



## **Cray C and C++ Reference Manual**

**S-2179-71**

---

© 1996-2000, 2002-2009 Cray Inc. All Rights Reserved. This document or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of Cray Inc.

---

#### U.S. GOVERNMENT RESTRICTED RIGHTS NOTICE

The Computer Software is delivered as "Commercial Computer Software" as defined in DFARS 48 CFR 252.227-7014.

All Computer Software and Computer Software Documentation acquired by or for the U.S. Government is provided with Restricted Rights. Use, duplication or disclosure by the U.S. Government is subject to the restrictions described in FAR 48 CFR 52.227-14 or DFARS 48 CFR 252.227-7014, as applicable.

Technical Data acquired by or for the U.S. Government, if any, is provided with Limited Rights. Use, duplication or disclosure by the U.S. Government is subject to the restrictions described in FAR 48 CFR 52.227-14 or DFARS 48 CFR 252.227-7013, as applicable.

---

Cray, LibSci, and UNICOS are federally registered trademarks and Active Manager, Cray Apprentice2, Cray Apprentice2 Desktop, Cray C++ Compiling System, Cray CX1, Cray Fortran Compiler, Cray Linux Environment, Cray SeaStar, Cray SeaStar2, Cray SeaStar2+, Cray SHMEM, Cray Threadstorm, Cray X1, Cray X1E, Cray X2, Cray XD1, Cray XMT, Cray XR1, Cray XT, Cray XT3, Cray XT4, Cray XT5, Cray XT5<sub>h</sub>, Cray XT5m, CrayDoc, CrayPort, CRInform, ECOphlex, Libsci, NodeKARE, RapidArray, UNICOS/lc, UNICOS/mk, and UNICOS/mp are trademarks of Cray Inc.

---

GNU is a trademark of The Free Software Foundation. ISO is a trademark of International Organization for Standardization (Organisation Internationale de Normalisation). O2 is a trademark of Silicon Graphics, Inc. SGI and Silicon Graphics are trademarks of Silicon Graphics, Inc. TotalView is a trademark of TotalView Technologies LLC. UNIX, the "X device," X Window System, and X/Open are trademarks of The Open Group in the United States and other countries. All other trademarks are the property of their respective owners.

---

The UNICOS, UNICOS/mk, and UNICOS/mp operating systems are derived from UNIX System V. These operating systems are also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

---

Portions of this document were copied by permission of OpenMP Architecture Review Board from OpenMP C and C++ Application Program Interface, Version 2.0, March 2002, Copyright © 1997-2002, OpenMP Architecture Review Board.

---

Version 2.0 Published January 1996 Original Printing. This manual supports the C and C++ compilers contained in the Cray C++ Programming Environment release 2.0. On all Cray systems, the C++ compiler is Cray C++ 2.0. On Cray systems with IEEE floating-point hardware, the C compiler is Cray Standard C 5.0. On Cray systems without IEEE floating-point hardware, the C compiler is Cray Standard C 4.0.

Version 3.0 Published May 1997 This rewrite supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.0, which is supported on all systems except the Cray T3D system. On all supported Cray systems, the C++ compiler is Cray C++ 3.0 and the C compiler is Cray C 6.0.

Version 3.0.2 Published March 1998 This para supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.0.2, which is supported on all systems except the Cray T3D system. On all supported Cray systems, the C++ compiler is Cray C++ 3.0.2 and the C compiler is Cray C 6.0.2.

Version 3.1 Published August 1998 This para supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.1, which is supported on all systems except the Cray T3D system. On all supported Cray systems, the C++ compiler is Cray C++ 3.1 and the C compiler is Cray C 6.1.

Version 3.2 Published January 1999 This para supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.2, which is supported on all systems except the Cray T3D system. On all supported Cray systems, the C++ compiler is Cray C++ 3.2 and the C compiler is Cray C 6.2.

Version 3.3 Published July 1999 This para supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.3, which is supported on the Cray SV1, Cray C90, Cray J90, and Cray T90 systems running UNICOS 10.0.0.5 and later, and Cray T3E systems running UNICOS/mk 2.0.4 and later. On all supported Cray systems, the C++ compiler is Cray C++ 3.3 and the C compiler is Cray C 6.3.

Version 3.4 Published August 2000 This para supports the Cray C 6.4 and Cray C++ 3.4 releases running on UNICOS and UNICOS/mk operating systems. It includes updates to para 3.3.

Version 3.4 Published October 2000 This para supports the Cray C 6.4 and Cray C++ 3.4 releases running on UNICOS and UNICOS/mk operating systems. This para supports a new inlining level, inline4.

Version 3.6 Published June 2002 This para supports the Cray Standard C 6.6 and Cray Standard C++ 3.6 releases running on UNICOS and UNICOS/mk operating systems.

Version 4.1 Published August 20, 2002 Draft version to support Cray C 7.1 and Cray C++ 4.1 releases running on UNICOS/mp operating systems.

Version 4.2 Published December 20, 2002 Draft version to support Cray C 7.2 and Cray C++ 4.2 releases running on UNICOS/mp operating systems.

Version 4.3 Published March 31, 2003 Draft version to support Cray C 7.3 and Cray C++ 4.3 releases running on UNICOS/mp operating systems.

Version 5.0 Published June 2003 Supports Cray C++ 5.0 and Cray C 8.0 releases running on UNICOS/mp 2.1 or later operating systems.

Version 5.1 Published October 2003 Supports Cray C++ 5.1 and Cray C 8.1 releases running on UNICOS/mp 2.2 or later operating systems.

Version 5.2 Published April 2004 Supports Cray C++ 5.2 and Cray C 8.2 releases running on UNICOS/mp 2.3 or later operating systems.

Version 5.3 Published November 2004 Supports Cray C++ 5.3 and Cray C 8.3 releases running on UNICOS/mp 2.5 or later operating systems.

Version 5.4 Published March 2005 Supports Cray C++ 5.4 and Cray C 8.4 releases running on UNICOS/mp 3.0 or later operating systems.

Version 5.5 Published December 2005 Supports Cray C++ 5.5 and Cray C 8.5 releases running on UNICOS/mp 3.0 or later operating systems.

Version 5.6 Published March 2007 Supports Cray C++ 5.6 and Cray C 8.6 releases running on Cray X1 series systems.

Version 6.0 Published September 2007 Supports the Cray C and Cray C++ 6.0 release running on Cray X1 series and Cray X2 systems.

Version 7.0 Published December 2008 Supports the Cray C and (Deferred implementation) C++ compilers running on Cray XT compute nodes.

Version 7.1 Published June 2009 Supports the Cray C and C++ compilers running on Cray XT compute nodes.

---



# New Features

## Support Cray C++ compiler

Documented support of the Cray C++ compiler ([Cray C++ Compiler on page 18](#)).

## Turn the insertion of CrayPat function entry points tracing calls on or off.

Documented support of the `-h [no]func_trace` option ([-h \[no\]func\\_trace on page 35](#)).

## Change the forwarding of aprun utility and resource limits

As of CLE 2.2, the aprun utility no longer forwards its user resource limits to each compute node, except for RLIMIT\_CORE and RLIMIT\_CPU ([Run Time Environment Variables on page 63](#)).

## Indicate a preference for threading or no threading

The `loop_info prefer_thread` and `loop_info prefer_nothread` directives indicate a preference for turning threading on or off for selected loops ([loop\\_info prefer\\_thread, prefer\\_nothread Directives on page 78](#)).

## Turn autothreading on or off

The `-h[no]autothread` option enables or disables automatic threading (see [-h \[no\]autothread on page 32](#)). The `autothread` and `noautothread` directives turn autothreading on and off for selected blocks of code ([autothread, noautothread Directives on page 73](#)).

## Turn off cache blocking

The `-h cache0` option turns off cache blocking, including directive-based blocking ([-h cachen on page 38](#)).

## Control the generation of DWARF information.

The `-h [no] dwarf` option controls whether DWARF debugging information is generated during compilation ([-h \[no\]dwarf on page 33](#)).

## Revise `-h list` documentation

Added documentation of the `-h list=d` option ([-h list on page 33](#)).

## Control the optimization of both OpenMP and automatic threading

The `-h threadn` option controls the optimization of OpenMP and autothreading ([-h threadn on page 36](#)).

Replace `-h smpn` option with `-h threadn`

See [-h threadn](#) on page 36.

Enable or disable the generation of threadsafe code

See [-h \[no\]threadsafe](#) on page 59.

Support AMD processors code named "shanghai" and "istanbul."

See [-h cpu=target\\_system](#) on page 57.

# Contents

---

	<i>Page</i>
<b>Introduction [1]</b>	<b>17</b>
1.1 General Compiler Description . . . . .	17
1.1.1 Cray C Compiler . . . . .	17
1.1.2 Cray C++ Compiler . . . . .	18
1.2 Related Publications . . . . .	18
<b>Invoking the C and C++ Compilers [2]</b>	<b>19</b>
2.1 CC Command . . . . .	20
2.2 cc Command . . . . .	20
2.3 Command Line Options . . . . .	21
2.4 Standard Language Conformance Options . . . . .	22
2.4.1 -h [no]c99 (cc) . . . . .	22
2.4.2 -h [no]conform(CC, cc), -h [no]stdc (cc) . . . . .	22
2.4.3 -h cfront (CC) . . . . .	23
2.4.4 -h [no]parse_templates (CC) . . . . .	23
2.4.5 -h [no]dep_name (CC) . . . . .	23
2.4.6 -h [no]exceptions (CC) . . . . .	23
2.4.7 -h [no]anachronisms (CC) . . . . .	23
2.4.8 -h new_for_init (CC) . . . . .	24
2.4.9 -h [no]tolerant (cc) . . . . .	24
2.4.10 -h [no]const_string_literals (CC) . . . . .	24
2.4.11 -h [no]gnu . . . . .	25
2.5 Template Language Options . . . . .	27
2.5.1 -h simple_templates (CC) . . . . .	27
2.5.2 -h [no]autoinstantiate (CC) . . . . .	27
2.5.3 -h one_instantiation_per_object (CC) . . . . .	27
2.5.4 -h instantiation_dir= <i>dirname</i> (CC) . . . . .	28
2.5.5 -h instantiate= <i>mode</i> (CC) . . . . .	28
2.5.6 -h [no]implicitinclude (CC) . . . . .	29
2.5.7 -h remove_instantiation_flags (CC) . . . . .	29

	<i>Page</i>
2.5.8 -hprelink_local_copy (CC)	29
2.5.9 -hprelink_copy_if_nonlocal (CC)	29
2.6 Virtual Function Options	29
2.6.1 -h forcevtbl (CC)	29
2.6.2 -h suppressvtbl (CC)	29
2.7 General Language Options	30
2.7.1 -h keep=file (CC)	30
2.7.2 -h restrict=args	30
2.7.3 -h [no]calchars	32
2.7.4 -h [no]signedshifts	32
2.8 General Optimization Options	32
2.8.1 -h [no]aggress	32
2.8.2 -h [no]autothread	32
2.8.3 -h display_opt	33
2.8.4 -h [no]dwarf	33
2.8.5 -h fusionn	33
2.8.6 -h [no]intrinsics	33
2.8.7 -h list	33
2.8.8 -h [no]msgs	35
2.8.9 -h [no]negmsgs	35
2.8.10 -h [no]omp_trace	35
2.8.11 -h [no]func_trace	35
2.8.12 -h [no]overindex	36
2.8.13 -h [no]pattern	36
2.8.14 -h profile_generate	36
2.8.15 -h threadn	36
2.8.16 -h unrolln	37
2.8.17 -O level	37
2.9 Automatic Cache Management Options	38
2.9.1 -h cachenn	38
2.10 Vector Optimization Options	39
2.10.1 -h vectorn	39
2.11 Inlining Optimization Options	40
2.11.1 -h ipan	41
2.11.2 -h ipafrom=source [source] ...	42
2.11.3 Combined Inlining	43
2.12 Scalar Optimization Options	43



	<i>Page</i>
2.12.1 -h [no]interchange . . . . .	43
2.12.2 -h scalarn . . . . .	44
2.12.3 -h [no]zeroinc . . . . .	44
2.13 Math Options . . . . .	44
2.13.1 -h fpn . . . . .	45
2.13.2 -h matherror . . . . .	47
2.14 Debugging Options . . . . .	47
2.14.1 -G <i>level</i> and -g . . . . .	47
2.14.2 -h [no]bounds (cc) . . . . .	48
2.14.3 -h dir_check . . . . .	48
2.14.4 -h zero . . . . .	48
2.15 Compiler Message Options . . . . .	48
2.15.1 -h msglevel_ <i>n</i> . . . . .	49
2.15.2 -h [no]message= <i>n</i> [: <i>n</i> ...] . . . . .	49
2.15.3 -h report= <i>args</i> . . . . .	49
2.15.4 -h [no]abort . . . . .	50
2.15.5 -h errorlimit . . . . .	50
2.16 Compilation Phase Options . . . . .	50
2.16.1 -E . . . . .	50
2.16.2 -P . . . . .	51
2.16.3 -h feonly . . . . .	51
2.16.4 -S . . . . .	51
2.16.5 -c . . . . .	51
2.16.6 -#, -##, and -### . . . . .	51
2.16.7 -W <i>phase</i> , " <i>opt</i> ..." . . . . .	51
2.16.8 -Y <i>phase</i> , <i>dirname</i> . . . . .	52
2.17 Preprocessing Options . . . . .	52
2.17.1 -C . . . . .	53
2.17.2 -D <i>macro</i> [=def] . . . . .	53
2.17.3 -h [no]pragma= <i>name</i> [: <i>name</i> ...] . . . . .	53
2.17.4 -I <i>includir</i> . . . . .	54
2.17.5 -M . . . . .	55
2.17.6 -nostdinc . . . . .	55
2.17.7 -U . . . . .	55
2.18 Loader Options . . . . .	55
2.18.1 -l <i>libname</i> . . . . .	56
2.18.2 -L <i>ldir</i> . . . . .	56

	<i>Page</i>
2.18.3 -o <i>outfile</i>	57
2.19 Miscellaneous Options	57
2.19.1 -h <i>cpu=target_system</i>	57
2.19.2 -h <i>ident=name</i>	57
2.19.3 -h <i>keepfiles</i>	57
2.19.4 -h <i>network=nice</i>	58
2.19.5 -h [no]omp	58
2.19.6 -h <i>prototype_intrinsics</i>	58
2.19.7 -h <i>taskn</i>	58
2.19.8 -h [no]threadsafe	59
2.19.9 -h <i>upc(cc)</i>	59
2.19.10 -V	59
2.19.11 -X <i>npes</i>	59
2.20 Command Line Examples	60
2.21 Compile Time Environment Variables	61
2.22 Run Time Environment Variables	63
2.23 OpenMP Environment Variables	63
<b>Using #pragma Directives [3]</b>	<b>65</b>
3.1 Protecting Directives	66
3.2 Directives in Cray C++	66
3.3 Loop Directives	66
3.4 Alternative Directive Form: <i>_Pragma</i>	67
3.5 General Directives	67
3.5.1 [no]bounds Directive	68
3.5.2 duplicate Directive	68
3.5.3 message Directive	70
3.5.4 cache Directive	71
3.5.5 cache_nt Directive	71
3.5.6 ident Directive	72
3.5.7 [no]opt Directive	72
3.5.8 autothread, noautothread Directives	73
3.5.9 Probability Directives	73
3.5.10 weak Directive	74
3.6 Instantiation Directives	76
3.7 Vectorization Directives	76
3.7.1 hand_tuned Directive	76
3.7.2 loop_info Directive	77

	<i>Page</i>
3.7.3 loop_info prefer_thread, prefer_nothread Directives . . . . .	78
3.7.4 nopattern Directive . . . . .	78
3.7.5 novector Directive . . . . .	79
3.7.6 permutation Directive . . . . .	79
3.7.7 [no]pipeline Directive . . . . .	80
3.7.8 prefetchvector Directive . . . . .	81
3.7.9 pgo_loop_info Directive . . . . .	81
3.7.10 safe_address Directive . . . . .	81
3.7.11 safe_conditional Directive . . . . .	83
3.7.12 shortloop and shortloop128 Directives . . . . .	84
3.8 Scalar Directives . . . . .	84
3.8.1 collapse and nocollapse Directives . . . . .	84
3.8.2 concurrent Directive . . . . .	85
3.8.3 interchange and nointerchange Directives . . . . .	86
3.8.4 noreduction Directive . . . . .	86
3.8.5 suppress Directive . . . . .	87
3.8.6 [no]unroll Directive . . . . .	87
3.8.7 [no]fusion Directive . . . . .	89
3.9 Inlining Directives . . . . .	89
3.9.1 clone_enable, clone_disable, clone_reset Directives . . . . .	90
3.9.2 inline_enable, inline_disable, and inline_reset Directives . . . . .	90
3.9.3 inline_always and inline_never Directives . . . . .	92
<b>Using OpenMP [4]</b>	<b>93</b>
4.1 Deferred OpenMP Features . . . . .	93
4.2 Cray Implementation Differences . . . . .	94
4.2.1 Pragmas . . . . .	94
4.2.1.1 atomic Construct . . . . .	94
4.2.1.2 for Construct . . . . .	94
4.2.1.3 parallel Construct . . . . .	94
4.2.1.4 private Clause . . . . .	95
4.2.1.5 threadprivate Construct . . . . .	95
4.2.2 OpenMP Library Routines . . . . .	95
4.2.2.1 omp_get_max_active_levels() . . . . .	95
4.2.2.2 omp_set_dynamic() . . . . .	95
4.2.2.3 omp_set_schedule() . . . . .	95
4.2.2.4 omp_set_max_active_levels() . . . . .	95
4.2.2.5 omp_set_nested() . . . . .	96

	<i>Page</i>
4.2.2.6 <code>omp_set_num_threads()</code> . . . . .	96
4.2.3 OpenMP Environment Variables . . . . .	96
4.2.3.1 <code>OMP_DYNAMIC</code> . . . . .	96
4.2.3.2 <code>OMP_MAX_ACTIVE_LEVELS</code> . . . . .	96
4.2.3.3 <code>OMP_NESTED</code> . . . . .	96
4.2.3.4 <code>OMP_NUM_THREADS</code> . . . . .	96
4.2.3.5 <code>OMP_SCHEDULE</code> . . . . .	96
4.2.3.6 <code>OMP_STACKSIZE</code> . . . . .	96
4.2.3.7 <code>OMP_THREAD_LIMIT</code> . . . . .	97
4.2.3.8 <code>OMP_WAIT_POLICY</code> . . . . .	97
4.3 Compiler Options Affecting OpenMP . . . . .	97
4.4 OpenMP Program Execution . . . . .	97
<b>Using Cray Unified Parallel C (UPC) [5]</b>	<b>99</b>
5.1 Cray Implementation Differences . . . . .	100
5.2 Compiling and Executing UPC Code . . . . .	100
<b>Using Cray C++ Libraries [6]</b>	<b>103</b>
6.1 Unsupported Standard C++ Library Features . . . . .	103
<b>Using Cray C++ Template Instantiation [7]</b>	<b>105</b>
7.1 Simple Instantiation . . . . .	106
7.2 Prelinker Instantiation . . . . .	107
7.3 Instantiation Modes . . . . .	110
7.4 One Instantiation Per Object File . . . . .	110
7.5 Instantiation <code>#pragma</code> Directives . . . . .	111
7.6 Implicit Inclusion . . . . .	112
<b>Using Cray C Extensions [8]</b>	<b>115</b>
8.1 Complex Data Extensions . . . . .	115
8.2 <code>fortran</code> Keyword . . . . .	116
8.3 Hexadecimal Floating-point Constants . . . . .	116
<b>Using Predefined Macros [9]</b>	<b>119</b>
9.1 Macros Required by the C and C++ Standards . . . . .	120
9.2 Macros Based on the Host Machine . . . . .	120
9.3 Macros Based on the Target Machine . . . . .	121
9.4 Macros Based on the Compiler . . . . .	121
9.5 UPC Predefined Macros . . . . .	122

	<i>Page</i>
<b>Running C and C++ Applications [10]</b>	<b>123</b>
<b>Debugging Cray C and C++ Code [11]</b>	<b>125</b>
11.1 TotalView Debugger . . . . .	125
11.2 Compiler Debugging Options . . . . .	126
<b>Using Interlanguage Communication [12]</b>	<b>127</b>
12.1 Calls between C and C++ Functions . . . . .	127
12.2 Calling Fortran Functions and Subroutines from C or C++ . . . . .	129
12.2.1 Requirements . . . . .	129
12.2.2 Argument Passing . . . . .	130
12.2.3 Array Storage . . . . .	130
12.2.4 Logical and Character Data . . . . .	131
12.2.5 Accessing Named Common from C and C++ . . . . .	131
12.2.6 Accessing Blank Common from C or C++ . . . . .	133
12.2.7 Cray C and Fortran Example . . . . .	135
12.2.8 Calling a Fortran Program from Cray C++ . . . . .	137
12.3 Calling a C or C++ Function from Fortran . . . . .	138
<b>Implementation-defined Behavior [13]</b>	<b>141</b>
13.1 Messages . . . . .	141
13.2 Environment . . . . .	141
13.2.1 Identifiers . . . . .	142
13.2.2 Types . . . . .	142
13.2.3 Characters . . . . .	143
13.2.4 Wide Characters . . . . .	144
13.2.5 Integers . . . . .	144
13.2.6 Arrays and Pointers . . . . .	144
13.2.7 Registers . . . . .	145
13.2.8 Classes, Structures, Unions, Enumerations, and Bit Fields . . . . .	145
13.2.9 Qualifiers . . . . .	146
13.2.10 Declarators . . . . .	146
13.2.11 Statements . . . . .	146
13.2.12 Exceptions . . . . .	146
13.2.13 System Function Calls . . . . .	146
13.3 Preprocessing . . . . .	146
<b>Appendix A Using Libraries and the Loader</b>	<b>149</b>
A.1 Cray C and C++ Libraries . . . . .	149

	<i>Page</i>
A.2 Loader . . . . .	149
<b>Appendix B Using Cray C and C++ Dialects</b>	<b>151</b>
B.1 C++ Language Conformance . . . . .	151
B.1.1 Unsupported C++ Language Features . . . . .	151
B.1.2 Supported C++ Language Features . . . . .	151
B.2 C++ Anachronisms Accepted . . . . .	155
B.3 Extensions Accepted in Normal C++ Mode . . . . .	156
B.4 Extensions Accepted in C or C++ Mode . . . . .	157
B.5 C++ Extensions Accepted in cfront Compatibility Mode . . . . .	159
<b>Appendix C Using the Compiler Message System</b>	<b>165</b>
C.1 Expanding Messages with the <code>explain</code> Command . . . . .	165
C.2 Controlling the Use of Messages . . . . .	165
C.2.1 Command Line Options . . . . .	166
C.2.2 Environment Options for Messages . . . . .	166
C.2.3 <code>ORIG_CMD_NAME</code> Environment Variable . . . . .	167
C.3 Message Severity . . . . .	167
C.4 Common System Messages . . . . .	169
<b>Appendix D Using Intrinsic Functions</b>	<b>171</b>
D.1 Bit Operations . . . . .	172
D.2 Mask Operations . . . . .	172
D.3 Miscellaneous Operations . . . . .	173
<b>Glossary</b>	<b>175</b>
<b>Tables</b>	
Table 1. GCC C Language Extensions . . . . .	25
Table 2. GCC C++ Language Extensions . . . . .	27
Table 3. <code>-h</code> Option Descriptions . . . . .	38
Table 4. Cache Levels . . . . .	39
Table 5. Automatic Inlining Specifications . . . . .	41
Table 6. File Types . . . . .	43
Table 7. Floating-point Optimization Levels . . . . .	46
Table 8. <code>-G level</code> Definitions . . . . .	47
Table 9. <code>-W phase</code> Definitions . . . . .	52
Table 10. <code>-Y phase</code> Definitions . . . . .	52
Table 11. <code>-h pragma</code> Directive Processing . . . . .	53
Table 12. Data Type Mapping . . . . .	142

	<i>Page</i>
Table 13. Packed Characters . . . . .	143
Table 14. Unrecognizable Escape Sequences . . . . .	143





# Introduction [1]

---

The Cray Compiling Environment (CCE) contains both the Cray C and C++ compilers. The Cray C compiler conforms to the International Organization of Standards (ISO) standard ISO/IEC 9899:1999 (C99). The Cray C++ compiler conforms to the ISO/IEC 14882:1998 standard, with some exceptions. The exceptions are noted in [Appendix B, Using Cray C and C++ Dialects on page 151](#).

You log in either to a Cray XT login node or a standalone application development system and use the Cray XT Environment and related products to create your executables. You run your executables on Cray XT compute nodes. For further information about Cray XT login nodes and user environment, see the *Cray XT Programming Environment User's Guide*. For further information about the standalone application development platform, see the *Cray Application Developer's Environment Installation Guide*.

Throughout this manual, the differences between the Cray C and C++ compilers are noted when appropriate. When there is no difference, the phrase *the compiler* refers to both compilers. All compiler command options apply to Cray C and C++ unless so noted.

## 1.1 General Compiler Description

Both the Cray C and C++ compilers are contained within the Cray Compiling Environment (CCE). If you are compiling code written in C, use the `cc` command to compile source files. If you are compiling code written in C++, use the `CC` command.

### 1.1.1 Cray C Compiler

The Cray C compiler consists of a preprocessor, a language parser, an optimizer, and a code generator. You invoke the Cray C compiler with the `cc` compiler driver command. The `cc` command is described in [cc Command on page 20](#). This command and its options are also described in the `craycc(1)` man page. See [Command Line Examples on page 60](#).

## 1.1.2 Cray C++ Compiler

The Cray C++ compiler consists of a preprocessor, a language parser, a prelinker, an optimizer, and a code generator. You invoke the Cray C++ compiler with the CC compiler driver command. The CC command is described in [CC Command on page 20](#) and the `crayCC(1)` man page. See [Command Line Examples on page 60](#).

## 1.2 Related Publications

The following documents contain additional information that may be helpful:

- `cc(1)` compiler driver man page for all Cray XT C compilers
- `craycc(1)` man page for the Cray C compiler
- `CC(1)` compiler driver man page for all Cray XT C++ compilers
- `crayCC(1)` man page for the Cray C++ compiler
- `intro_pragmas(1)` man page
- *Cray Fortran Reference Manual*
- `ftn(1)` compiler driver man page for all Cray XT Fortran compilers
- `crayftn(1)` man page for the Cray Fortran compiler
- *Cray XT Programming Environment User's Guide*
- `aprun(1)` man page
- *Using Cray Performance Analysis Tools*
- *Cray Application Developer's Environment Installation Guide*

# Invoking the C and C++ Compilers [2]

---

This chapter describes the compiler driver commands that you use to launch the Cray C and C++ compilers. The following commands invoke the compilers:

- `CC`, which invokes the Cray C++ compiler.
- `cc`, which invokes the Cray C compiler.
- `cpp`, the C language preprocessor, is not part of the Cray Compilation Environment (CCE). The `cpp` command resolves to the GNU `cpp` command and does not predefine any Cray compiler-specific macros ([Chapter 9, Using Predefined Macros on page 119](#)). If the predefinition of the Cray compiler-specific macros is required, then use the `cc` or `CC` command to do the source preprocessing using the `-E` or `-P` option.

A successful compilation creates an executable, named `a.out` by default, that reflects the contents of the source code and any referenced library functions. You use the `aprun` command to run the executable on the Cray XT compute nodes.

For example, the following command sequence compiles file `mysource.c` and launches the resulting executable program on 64 compute nodes:

```
% cc mysource.c
% aprun -n 64 ./a.out
```

With the use of appropriate options, you can direct the compiler to generate intermediate translations, including relocatable object files (`-c` option), assembly source expansions (`-S` option), or the output of the preprocessor phase of the compiler (`-P` or `-E` option). In general, you can save the intermediate files and reference them later on a `CC` or `cc` command, with other files or libraries included as necessary.

By default, the `CC` and `cc` commands automatically call the loader, which creates an executable file. If only one source file is specified, the object file (`*.o`) is deleted. If more than one source file is specified, the object files are retained.

For example, the following command creates and retains object files `file1.o`, `file2.o`, and `file3.o`, and creates the executable file `a.out`:

```
% cc file1.c file2.c file3.c
```

The following command creates `file.o` and `a.out`; `file.o` is not retained.

```
% cc file.c
```

## 2.1 cc Command

The `CC` command invokes the Cray C++ compiler. The `CC` command accepts C++ source files with the following suffixes:

```
.c
.C
.i
.c++
.C++
.cc
.cxx
.Cxx
.CXX
.CC
```

The `.i` files are created when the preprocessing compiler command option (`-P`) is used. The `CC` command also accepts object files with the `.o` suffix, library files with the `.a` suffix, and assembler source files with the `.s` suffix.

The `CC` command format is as follows:

```
CC [-c] [-C] [-D macro[=def]] [-E] [-g] [-G level]
[-h arg] [-I incldir] [-l libfile] [-L ldir] [-M] [-nostdinc]
[-o outfile] [-O level] [-P] [-S] [-U macro] [-V]
[-Wphase, "opt ..."] [-X npes] [-Y phase, dirname] [-#] [-##] [-###]
files ...
```

For an explanation of the command line options, see [Command Line Options on page 21](#).

## 2.2 cc Command

The `cc` command invokes the Cray C compiler. The `cc` command accepts C source files that have the `.c` and `.i` suffixes; object files with the `.o` suffix; library files with the `.a` suffix; and assembler source files with the `.s` suffix.

The `cc` command format is as follows:

```
cc [-c] [-C] [-D macro[=def]] [-E] [-g] [-G level]
[-h arg] [-I incldir] [-l libfile] [-L ldir] [-M] [-nostdinc]
[-o outfile] [-O level] [-P] [-S] [-U macro] [-V]
[-W phase, "opt ..."] [-X npes] [-Y phase, dirname] [-#] [-##] [-###]
files ...
```

For an explanation of the command line options, see [Command Line Options on page 21](#).

## 2.3 Command Line Options

The following subsections describe options for the `CC` and `cc` commands. These options are grouped according to the following functions:

- Standard conformance options ([Standard Language Conformance Options on page 22](#))
- Template language options ([Template Language Options on page 27](#))
- Virtual function options ([Virtual Function Options on page 29](#))
- General language options ([General Language Options on page 30](#))
- General optimization options ([General Optimization Options on page 32](#))
- Automatic cache management options (`-h cachem` on page 38)
- Vector optimization options ([Vector Optimization Options on page 39](#))
- Inlining options ([Inlining Optimization Options on page 40](#))
- Scalar optimization options ([Scalar Optimization Options on page 43](#))
- Math options ([Math Options on page 44](#))
- Debugging options ([Debugging Options on page 47](#))
- Compiler message options ([Compiler Message Options on page 48](#))
- Compilation phase options ([Compilation Phase Options on page 50](#))
- Preprocessing options ([Preprocessing Options on page 52](#))
- Loader options ([Loader Options on page 55](#))
- Miscellaneous options ([Miscellaneous Options on page 57](#))
- Command line examples ([Command Line Examples on page 60](#))

Options other than those described in this manual are passed to the loader.

There are many options that start with `-h`. You can specify multiple `-h` options using commas to separate the arguments. For example, the `-h parse_templates` and `-h fp0` command line options can be specified as `-h parse_templates,fp0`.

If you specify conflicting options, the option specified last on the command line overrides the previously specified option. Exceptions to this rule are noted in the individual descriptions of the options.

The following examples illustrate the use of conflicting options:

- In this example, `-h fp0` overrides `-h fp1`:  

```
% cc -h fp1,fp0 myfile.c
```
- In this example, `-h vector2` overrides the earlier vector optimization level 3 implied by the `-O3` option:  

```
% CC -O3 -h vector2 myfile.C
```

Most `#pragma` directives override corresponding command line options. Exceptions to this rule are noted in descriptions of options or `#pragma` directives.

## 2.4 Standard Language Conformance Options

This section describes standard conformance language options. Each subsection heading shows in parentheses the compiler with which the option can be used.

### 2.4.1 `-h [no]c99 (cc)`

Default: `-h c99`

This option enables or disables language features new to the C99 standard and Cray C compiler, while providing support for features that were previously defined as Cray extensions. If the previous implementation of the Cray extension differed from the C99 standard, both implementations will be available when the `-h c99` option is enabled. The `-h c99` option is also required for C99 features not previously supported as extensions.

When `-h noc99` is used, C99 language features such as variable-length arrays (VLAs) and restricted pointers that were available as extensions previously to adoption of the C99 standard remain available to you.

### 2.4.2 `-h [no]conform (CC, cc), -h [no]stdc (cc)`

Default: `-h noconform, -h nostdc`

The `-h conform` and `-h stdc` options specify strict conformance to the ISO C standard or the ISO C++ standard. The `-h noconform` and `-h nostdc` options specify partial conformance to the standard. The `-h exceptions`, `-h dep_name`, `-h parse_templates`, and `-h const_string_literals` options are enabled by the `-h conform` option in Cray C++.

### 2.4.3 -h cfront (CC)

The `-h cfront` option causes the Cray C++ compiler to accept or reject constructs that were accepted by previous `cfront`-based compilers (such as Cray C++ 1.0) but which are not accepted in the C++ standard. The `-h anachronisms` option is implied when `-h cfront` is specified.

### 2.4.4 -h [no]parse\_templates (CC)

Default: `-h noparse_templates`

This option allows existing code that defines templates using previous versions of the Cray Standard Template Library (STL) (before Programming Environment 3.6) to compile successfully with the `-h conform` option. Consequently, this allows you to compile existing code without having to use the Cray C++ STL. To do this, use the `noparse_templates` option. Also, the compiler defaults to this mode when the `-h dep_name` option is used. To have the compiler verify that your code uses the Cray C++ STL properly, use the `parse_templates` option.

### 2.4.5 -h [no]dep\_name (CC)

Default: `-h nodep_name`

This option enables or disables dependent name processing (that is, the separate lookup of names in templates when the template is parsed and when it is instantiated). The `-h dep_name` option cannot be used with the `-h noparse_templates` option.

### 2.4.6 -h [no]exceptions (CC)

Default: The default is `-h exceptions`; however, if the `CRAYOLDCPPLIB` environment variable is set to a nonzero value, the default is `-h noexceptions`.

The `-h exceptions` option enables support for exception handling. The `-h noexceptions` option issues an error whenever an exception construct, a `try` block, a `throw` expression, or a `throw` specification on a function declaration is encountered. The `-h exceptions` option is enabled by `-h conform`.

### 2.4.7 -h [no]anachronisms (CC)

Default: `-h noanachronisms`

The `-h [no]anachronisms` option disables or enables anachronisms in Cray C++. This option is overridden by `-h conform`.

## 2.4.8 -h new\_for\_init (CC)

The `-h new_for_init` option enables the new scoping rules for a declaration in a *for-init-statement*. This means that the new standard-conforming rules are in effect; the entire `for` statement is wrapped in its own implicitly generated scope. The `-h new_for_init` option is implied by the `-h conform` option.

This is the result of the scoping rule:

```
{
    .
    .
    .
    for (int i = 0; i < n; i++) {
        .
        .
        .
    } // scope of i ends here for -h new_for_init
    .
    .
    .
} // scope of i ends here by default
```

## 2.4.9 -h [no]tolerant (cc)

Default:        `-h notolerant`

The `-h tolerant` option allows older, less standard C constructs, thereby making it easier to port code written for previous C compilers. Errors involving comparisons or assignments of pointers and integers become warnings. The compiler generates casts so that the types agree. With `-h notolerant`, the compiler is intolerant of the older constructs.

The `-h tolerant` option causes the compiler to tolerate accessing an object with one type through a pointer to an entirely different type. For example, a pointer to a `long` might be used to access an object declared with type `double`. Such references violate the C standard and should be eliminated if possible. They can reduce the effectiveness of alias analysis and inhibit optimization.

## 2.4.10 -h [no]const\_string\_literals (CC)

Default:        `-h noconst_string_literals`

The `-h [no]const_string_literals` option controls whether string literals are `const` (as required by the standard) or `non-const` (as was true in earlier versions of the C++ language).



## 2.4.11 -h [no]gnu

Default:        -h nognu

The `-h gnu` option enables the compiler to recognize the subset of the GCC version 3.3.2 extensions to C listed in Table 1. Table 2 lists the extensions that apply only to C++.

For detailed descriptions of the GCC C and C++ language extensions, see <http://gcc.gnu.org/onlinedocs/>.

**Table 1. GCC C Language Extensions**

<b>GCC C Language Extension</b>	<b>Description</b>
Typeof	<code>typeof</code> : referring to the type of an expression
Lvalues	Using <code>? :</code> , and casts in lvalues
Conditionals	Omitting the middle operand of a <code>? :</code> expression
Long Long	Double-word integers -- <code>long long int</code>
Complex	Data types for complex numbers
Statement Exprs	Putting statements and declarations inside expressions
Zero Length	Zero-length arrays
Variable Length	Arrays whose length is computed at run time
Empty Structures	Structures with no members; applies to C but not C++
Variadic Macros	Macros with a variable number of arguments
Escaped Newlines	Slightly looser rules for escaped newlines
Multiline strings	String literals with embedded newlines
Initializers	Non-constant initializers
Compound Literals	Compound literals give structures, unions or arrays as values
Designated Inits	Labeling elements of initializers
Cast to Union	Casting to union type from any member of the union
Case Ranges	<code>'case 1 ... 9'</code> and such
Mixed Declarations	Mixing declarations and code
Attribute Syntax	Formal syntax for attributes
Function Prototypes	Prototype declarations and old-style definitions; applies to C but not C++
C++ Comments	C++ comments are recognized
Dollar Signs	Dollar sign is allowed in identifiers
Character Escapes	<code>\e</code> stands for the character <code>&lt;ESC&gt;</code>
Alignment	Inquiring about the alignment of a type or variable

<b>GCC C Language Extension</b>	<b>Description</b>
Inline	Defining inline functions (as fast as macros)
Alternate Keywords	<code>__const__</code> , <code>__asm__</code> , and so on, for header files
Incomplete Enums	<code>enum foo;</code> , with details to follow
Function Names	Printable strings which are the name of the current function
Return Address	Getting the return or frame address of a function
Unnamed Fields	Unnamed struct/union fields within structs/unions
Function Attributes: <code>nothrow</code> ; <code>format</code> , <code>format_arg</code> ; <code>deprecated</code> ; <code>used</code> ; <code>unused</code> ; <code>alias</code> ; <code>weak</code>	Declaring that functions have no side effects, or that they can never return
Variable Attributes: <code>alias</code> ; <code>deprecated</code> ; <code>unused</code> ; <code>used</code> ; <code>transparent_union</code> ; <code>weak</code> ;	Specifying attributes of variables
Type Attributes: <code>deprecated</code> ; <code>unused</code> ; <code>used</code> ; <code>transparent_union</code>	Specifying attributes of types
Asm Labels	Specifying the assembler name to use for a C symbol
Other Builtins:	Other built-in functions
<code>__builtin_types_compatible_p</code> , <code>__builtin_choose_expr</code> , <code>__builtin_constant_p</code> , <code>__builtin_huge_val</code> , <code>__builtin_huge_valf</code> , <code>__builtin_huge_vall</code> , <code>__builtin_inf</code> , <code>__builtin_inff</code> , <code>__builtin_infl</code> , <code>__builtin_nan</code> , <code>__builtin_nanf</code> , <code>__builtin_nanl</code> , <code>__builtin_nans</code> , <code>__builtin_nansf</code> , <code>__builtin_nansl</code>	

Special files such as `/dev/null` may be used as source files.

The supported subset of the GCC version 3.3.2 extensions to C++ are listed in [Table 2](#).

**Table 2. GCC C++ Language Extensions**

<b>GCC C++ Extensions</b>	<b>Description</b>
Min and Max	C++ minimum and maximum operators
Restricted Pointers	C99 restricted pointers and references
Backwards Compatibility	Compatibilities with earlier definitions of C++
Strong Using	A <code>using</code> directive with <code>__attribute__((strong))</code>
Explicit template specializations	Attributes may be used on explicit template specializations

## 2.5 Template Language Options

This section describes template language options. For more information about template instantiation, see [Chapter 7, Using Cray C++ Template Instantiation on page 105](#). Each subsection heading shows in parentheses the compiler with which the option can be used.

### 2.5.1 `-h simple_templates` (CC)

The `-h simple_templates` option enables simple template instantiation by the Cray C++ compiler. For more information about template instantiation, see [Chapter 7, Using Cray C++ Template Instantiation on page 105](#).

### 2.5.2 `-h [no]autoinstantiate` (CC)

Default: `-h noautoinstantiate`

The `-h [no]autoinstantiate` option enables or disables prelinker (automatic) instantiation of templates by the Cray C++ compiler. For more information about template instantiation, see [Chapter 7, Using Cray C++ Template Instantiation on page 105](#).

### 2.5.3 `-h one_instantiation_per_object` (CC)

The `-h one_instantiation_per_object` option puts each template instantiation used in a compilation into a separate object file that has a `.int.o` extension. The primary object file will contain everything else that is not an instantiation. For the location of the object files, see the `-h instantiation_dir` option.

## 2.5.4 -h instantiation\_dir=*dirname* (CC)

The `-h instantiation_dir=dirname` option specifies the instantiation directory that the `-h one_instantiation_per_object` option should use. If directory *dirname* does not exist, it will be created. The default directory is `./Template.dir`.

## 2.5.5 -h instantiate=*mode* (CC)

Default: `-h instantiate=none`

Usually, during compilation of a source file, no template entities are instantiated (except those assigned to the file by automatic instantiation). However, you can change the overall instantiation mode by using the `-h instantiate=mode` option, where *mode* is specified as `none`, `used`, `all`, or `local`. The default is `-h instantiate=none`. To change the overall instantiation mode, specify one of the following for *mode*:

<code>none</code>	Default. Does not automatically create instantiations of any template entities. This is the most appropriate mode when prelinker (automatic) instantiation is enabled.
<code>used</code>	Instantiates only those template entities that were used in the compilation. This includes all static data members that have template definitions.
<code>all</code>	Instantiates all template functions declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members are instantiated, regardless of whether they were used. Nonmember template functions are instantiated even if the only reference was a declaration.
<code>local</code>	Similar to <code>instantiate=used</code> , except that the functions are given internal linkage. This mode provides a simple mechanism for those who are not familiar with templates. The compiler instantiates the functions used in each compilation unit as local functions, and the program links and runs correctly (barring problems due to multiple copies of local static variables). This mode may generate multiple copies of the instantiated functions and is not suitable for production use. This mode cannot be used in conjunction with prelinker (automatic) template instantiation. Automatic template instantiation is disabled by this mode.

If CC is given a single source file to compile and link, all instantiations are done in the single source file and, by default, the `instantiate=used` mode is used to suppress prelinker instantiation.

## 2.5.6 `-h [no]implicitinclude (CC)`

Default: `-h noimplicitinclude`

The `-h [no]implicitinclude` option enables or disables implicit inclusion of source files as a method of finding definitions of template entities to be instantiated.

## 2.5.7 `-h remove_instantiation_flags (CC)`

The `-h remove_instantiation_flags` option causes the prelinker to recompile all the source files to remove all instantiation flags.

## 2.5.8 `-h prelink_local_copy (CC)`

The `-h prelink_local_copy` indicates that only local files (for example, files in the current directory) are candidates for assignment of instantiations.

## 2.5.9 `-h prelink_copy_if_nonlocal (CC)`

The `-h prelink_copy_if_nonlocal` option specifies that assignment of an instantiation to a nonlocal object file will result in the object file being recompiled in the current directory.

# 2.6 Virtual Function Options

This section describes general language options.

## 2.6.1 `-h forcevtbl (CC)`

The `-h forcevtbl` option forces the definition of virtual function tables in cases where the heuristic methods used by the compiler to decide on definition of virtual function tables provide no guidance. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline, non-pure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table (but to define it as a local static entity). The `-h forcevtbl` option differs from the default behavior in that it does not force the definition to be local.

## 2.6.2 `-h suppressvtbl (CC)`

The `-h suppressvtbl` option suppresses the definition of virtual function tables in cases where the heuristic methods used by the compiler to decide on definition of virtual function tables provide no guidance.

## 2.7 General Language Options

This section describes general language options. Each subsection heading shows in parentheses the compiler with which the option can be used.

### 2.7.1 `-h keep=file (CC)`

When the `-h keep=file` option is specified, the static constructor/destructor object (`.o`) file is retained as *file*. This option is useful when linking `.o` files on a system that does not have a C++ compiler. The use of this option requires that the `main` function must be compiled by C++ and the static constructor/destructor function must be included in the link. With these precautions, mixed object files (files with `.o` suffixes) from C and C++ compilations can be linked into executables by using the loader command instead of the `CC` command.

### 2.7.2 `-h restrict=args`



The `-h restrict=args` option globally tells the compiler to treat certain classes of pointers as restricted pointers. You can use this option to enhance optimizations (this includes vectorization).

Classes of affected pointers are determined by the value contained in *args*, as follows:

<i>args</i>	Description
a	All pointers to object and incomplete types are considered restricted pointers, regardless of where they appear in the source code. This includes pointers in <code>class</code> , <code>struct</code> , and <code>union</code> declarations, type casts, function prototypes, and so on.



**Caution:** Do not specify `restrict=a` if, during execution of any function, an object is modified and that object is referenced through either two different pointers or through the declared name of the object and a pointer. Undefined behavior may result.

<i>args</i>	Description
<code>f</code>	<p>All function parameters that are pointers to objects or incomplete types can be treated as restricted pointers.</p> <p> <b>Caution:</b> Do not specify <code>restrict=f</code> if, during execution of any function, an object is modified and that object is referenced through either two different pointer function parameters or through the declared name of the object and a pointer function parameter. Undefined behavior may result.</p>
<code>t</code>	<p>All parameters that are <code>this</code> pointers can be treated as restricted pointers (Cray C++ only).</p> <p> <b>Caution:</b> Do not specify <code>restrict=t</code> if, during execution of any function, an object is modified and that object is referenced through the declared name of the object and a <code>this</code> pointer. Undefined behavior may result.</p>

The *args* arguments tell the compiler to assume that, in the current compilation unit, each pointer (`=a`), each pointer that is a function parameter (`=f`), or each `this` pointer (`=t`) points to a unique object. This assumption eliminates those pointers as sources of potential aliasing, and may allow additional vectorization or other optimizations. These options cause only data dependencies from pointer aliasing to be ignored, rather than all data dependencies.



**Caution:** The arguments make assertions about your program that, if incorrect, can introduce undefined behavior. You should not use `-h restrict=a` if, during the execution of any function, an object is modified and that object is referenced through either of the following:

- Two different pointers
- The declared name of the object and a pointer

The `-h restrict=f` and `-h restrict=t` options are subject to the analogous restriction, with "function parameter pointer" replacing "pointer."

### 2.7.3 -h [no]calchars

Default:        -h nocalchars

The `-h calchars` option allows the use of the `$` character in identifier names. This option is useful for porting code in which identifiers include this character. With `-h nocalchars`, this character is not allowed in identifier names.



**Caution:** Use this option with extreme care, because identifiers with this character are within CNL name space and are included in many library identifiers, internal compiler labels, objects, and functions. You must prevent conflicts between any of these uses, current or future, and identifier declarations or references in your code; any such conflict is an error.

### 2.7.4 -h [no]signedshifts

Default:        -h nosignedshifts

The `-h [no]signedshifts` option affects the result of the right shift operator. For the expression `e1 >> e2`, where `e1` has a signed type, when `-h signedshifts` is in effect, the vacated bits are filled with the sign bit of `e1`. When `-h nosignedshifts` is in effect, the vacated bits are filled with zeros, identical to the behavior when `e1` has an unsigned type.

Also, see [Integers on page 144](#) about the effects of this option when shifting integers.

## 2.8 General Optimization Options

This section describes general optimization options. Each subsection heading shows in parentheses the compiler with which the option can be used.

### 2.8.1 -h [no]aggress

Default:        -h noaggress

The `-h aggress` option provides greater opportunity to optimize loops that would otherwise be inhibited from optimization due to an internal compiler size limitation. `-h noaggress` leaves this size limitation in effect.

With `-h aggress`, internal compiler tables are expanded to accommodate larger loop bodies. This option can increase the compilation's time and memory size.

### 2.8.2 -h [no]autothread

Default:        -h noautothread

The `-h [no]autothread` option enables or disables automatic threading.



### 2.8.3 -h display\_opt

The `-h display_opt` option displays the current optimization settings for this compilation.

### 2.8.4 -h [no]dwarf

The `-h [no]dwarf` option controls whether DWARF debugging information is generated during compilation.

Default: `-h dwarf`

### 2.8.5 -h fusion*n*

Default: `-h fusion2`

The `-h fusionn` option controls loop fusion and changes the assertiveness of the `fusion` pragma. Loop fusion can improve the performance of loops, although in rare cases it may degrade performance. The *n* argument allows you to turn loop fusion on or off and determine where fusion should occur.

**Note:** Loop fusion is disabled when the scalar level is set to 0.

Default: `-h fusion2`

The values for *n* are:

- |   |   |
|---|---|
| 0 | No fusion (ignore all <code>fusion</code> pragmas and do not attempt to fuse other loops)                                   |
| 1 | Attempt to fuse loops that are marked by the <code>fusion</code> pragma.  |
| 2 | Attempt to fuse all loops (includes array syntax implied loops), except those marked with the <code>nofusion</code> pragma. |

### 2.8.6 -h [no]intrinsic

Default: `-h intrinsic`

The `-h intrinsic` option allows the use of intrinsic hardware functions, which allow direct access to some hardware instructions or generate inline code for some functions. This option has no effect on specially handled library functions.

Intrinsic functions are described in [Appendix D, Using Intrinsic Functions on page 171](#).

### 2.8.7 -h list

The `-h list=opt` option allows you to create listings and control their formats. The listings are written to `source_file_name_without_suffix.lst`.

The values for *opt* are:

- a Use all list options; *source\_file\_name\_without\_suffix.lst* includes a summary report, an options report, and the source listing.
- d Decompiles (translates) the intermediate representation of the compiler into listings that resemble the format of the source code. This is performed twice, resulting in two output files, at different points during the optimization process. You can use these files to examine the restructuring and optimization changes made by the compiler, which can lead to insights about changes you can make to your source code to improve its performance.

The compiler produces two decompilation listing files with these extensions per specified source file: *.opt* and *.cg*. The compiler generates the *.opt* file after applying most high-level loop nest transformations to the code. The code structure of this listing most resembles your source code and is readable by most users. In some cases, because of optimizations, the structure of the loops and conditionals will be significantly different than the structure in your source file.

The *.cg* file contains a much lower level of decompilation. It is quite close to what will be produced as assembly output. This version displays the intermediate text after all vector translation and other optimizations have been performed. An intimate knowledge of the hardware architecture of the system is helpful to understanding this listing.

The *.opt* and *.cg* files are intended as a tool for performance analysis and are not valid source code. The format and contents of the files can be expected to change from release to release.

- e Expand include files.  
**Note:** Using this option may result in a very large listing file. All system include files are also expanded.
- i Intersperse optimization messages within the source listing rather than at the end.
- m Create loopmark listing; *source\_file\_name\_without\_suffix.lst* includes summary report and source listing.
- s Create a complete source listing (include files not expanded).

Using `-h list=m` creates a loopmark listing. The *e*, *i*, *s*, and *w* options provide additional listing features. Using `-h list=a` combines all options.

## 2.8.8 -h [no]msgs

Default:        -h nomsgs

The `-h msgs` option causes the compiler to write optimization messages to `stderr`.

When the `-h msgs` option is in effect, you may request that a listing be produced so that you can see the optimization messages in the listing. For information about obtaining listings, see [-h list on page 33](#).

## 2.8.9 -h [no]negmsgs

Default:        -h nonegmsgs

The `-h negmsgs` option causes the compiler to generate messages to `stderr` that indicate why optimizations such as vectorization or inlining did not occur in a given instance.

The `-h negmsgs` option enables the `-h msgs` option. The `-h list=a` option enables the `-h negmsgs` option.

## 2.8.10 -h [no]omp\_trace

Default:        -h noomp\_trace (tracing is off)

The `-h [no]omp_trace` option turns the insertion of the CrayPat OpenMP tracing calls on or off.

## 2.8.11 -h [no]func\_trace

The `-h func_trace` option is for use only with CrayPat. If this option is specified, the compiler inserts CrayPat trace entry points into each function in the compiled source file. The names of the trace entry points are:

- `__pat_tp_func_entry`
- `__pat_tp_func_return`

These are resolved by CrayPat when the program is instrumented using the `pat_build` command. When the instrumented program is executed and it encounters either of these trace entry points, CrayPat captures the address of the current function and its return address.

## 2.8.12 -h [no]overindex

Default:        -h nooverindex

The `-h overindex` option declares that there are array subscripts that index a dimension of an array that is outside the declared bounds of that array. The `-h nooverindex` option declares that there are no array subscripts that index a dimension of an array that is outside the declared bounds of that array.

## 2.8.13 -h [no]pattern

Default:        -h pattern

The `-h [no]pattern` option globally enables or disables pattern matching.

When the compiler recognizes certain patterns in the source code, it replaces the construct with a call to an optimized library routine. A loop or statement that has been pattern matched and replaced with a call to a library routine is indicated with an A in the loopmark listing.

**Note:** Pattern matching is not always worthwhile. If there is a small amount of work in the pattern-matched construct, the call overhead may outweigh the time saved by using the optimized library routine. When compiling using the default optimization settings, the compiler attempts to determine whether each given candidate for pattern matching will in fact yield improved performance.

## 2.8.14 -h profile\_generate

The `-h profile_generate` option directs that the source code be instrumented for gathering profile information. The compiler inserts calls and data-gathering instructions to allow CrayPat to gather information about the loops in a compilation unit. If you use this option, you must run CrayPat on the resulting executable so the CrayPat data-gathering routines are linked in. For information about CrayPat and profile information, see the *Using Cray Performance Analysis Tools* guide.

## 2.8.15 -h threadn

Default:        -h thread2

The `-h threadn` options control the optimization of both OpenMP and automatic threading.

The values of *n* are:

- |   |   |
|---|---|
| 0 | No autothreading or OMP (OpenMP) threading.   |
| 1 | No parallel region expansion, no loop restructuring for OMP loops, no optimization across OMP constructs. |

- |   |   |
|---|---|
| 2 | Parallel region expansion, limited loop restructuring, optimization across OMP constructs.  |
| 3 | Reduction results may not be repeatable. Loop restructuring, including modifying iteration space for static schedules (breaking standard compliance). |

## 2.8.16 `-h unrolln`

Default: `-h unroll2`

The `-h unrolln` option globally controls loop unrolling and changes the assertiveness of the `unroll` pragma. By default, the compiler attempts to unroll all loops, unless the `nounroll` pragma is specified for a loop. Generally, unrolling loops increases single processor performance at the cost of increased compile time and code size.

The *n* argument allows you to turn loop unrolling on or off and specify where unrolling should occur. It also affects the assertiveness of the `unroll` pragma.

The values for *n* are:

- |   |   |
|---|---|
| 0 | No unrolling (ignore all <code>unroll</code> pragmas and do not attempt to unroll other loops).   |
| 1 | Attempt to unroll loops that are marked by the <code>unroll</code> pragma.  |
| 2 | Unroll loops when performance is expected to improve. Loops marked with the <code>unroll</code> or <code>nounroll</code> pragma override automatic unrolling. |

**Note:** Loop unrolling is disabled when the scalar level is set to 0.

## 2.8.17 `-O level`

Default: Equivalent to the appropriate `-h` option except that `-O3` is equivalent to `-h cache2`

The `-O level` option specifies the optimization level for a group of compiler features. Specifying `-O` with no argument is the same as not specifying the `-O` option; this syntax is supported for compatibility with other vendors.

A value of 0, 1, 2, or 3 sets that level of optimization for each of the `-h scalarn` and `-h vectorn` options.

The `-O` values of 0, 1, 2, or 3 set that level of optimization for `-h cachen` options, except that `-O3` is equivalent to `-h cache2`.

The `-O2` option is equivalent to `ipa2`, `scalar2`, `vector2`, `cache2`, and `thread2`.

Optimization features specified by `-O` are equivalent to the `-h` options listed in [Table 3](#).

**Table 3. `-h` Option Descriptions**

<b><code>-h</code> option</b>	<b>Description location</b>
<code>-h cachem</code>	<a href="#">-h cachem on page 38</a>
<code>-h vectorm</code>	<a href="#">-h vectorm on page 39</a>
<code>-h scalarm</code>	<a href="#">-h scalarm on page 44</a>

[Table 4](#) shows the equivalent level of automatic cache optimization for the `-h` option.

## 2.9 Automatic Cache Management Options

This section describes the automatic cache management options. Automatic cache management can be overridden by the use of the cache directives (`cache`, `cache_nt`, and `loop_info`).

### 2.9.1 `-h cachem`

Default: `-h cache2`

The `-h cachem` option specifies the levels of automatic cache management to perform. The default is `-h cache2`.

The values for *n* are:

- |   |   |
|---|---|
| 0 | Cache blocking (including directive-based blocking) is turned off. This level is compatible with all scalar and vector optimization levels.   |
| 1 | Conservative automatic cache management. Characteristics include moderate compile time. Symbols are placed in the cache when the possibility of cache reuse exists and the predicted cache footprint of the symbol in isolation is small enough to experience the reuse.  |
| 2 | Moderately aggressive automatic cache management. Characteristics include moderate compile time. Symbols are placed in the cache when the possibility of cache reuse exists and the predicted state of the cache model is such that the symbol will experience the reuse. |
| 3 | Aggressive automatic cache management. Characteristics include potentially high compile time. Symbols are placed in the cache when the possibility of cache reuse exists and the allocation of the symbol to the cache is predicted to increase the number of cache hits. |

**Table 4. Cache Levels**

<b>-O Option</b>	<b>Cache Level</b>
-O0	-h cache0
-O1	-h cache1
-O2	-h cache2
-O3	-h cache2

## 2.10 Vector Optimization Options

This section describes vector optimization options. Each subsection heading shows in parentheses the compiler command with which the option can be used.

### 2.10.1 `-h vectorn`

Default: `-h vector2`

The `-h vectorn` option specifies the level of automatic vectorizing to be performed. Vectorization results in significant performance improvements with a small increase in object code size. Vectorization directives are unaffected by this option.

The values of  $n$  are:

<u><math>n</math></u>	<u>Description</u>
0	No automatic vectorization. Characteristics include low compile time and small compile size. This option is compatible with all scalar optimization levels.
1	Specifies conservative vectorization. Characteristics include moderate compile time and size. No loop nests are restructured; only inner loops are vectorized. No vectorizations that might create false exceptions are performed. Results may differ slightly from results obtained when <code>-h vector0</code> is specified because of vector reductions.  The <code>-h vector1</code> option is compatible with <code>-h scalar1</code> , <code>-h scalar2</code> , and <code>-h scalar3</code> .
2	Specifies moderate vectorization. Characteristics include moderate compile time and size. Loop nests are restructured.  The <code>-h vector2</code> option is compatible with <code>-h scalar2</code> and <code>-h scalar3</code> .
3	Specifies aggressive vectorization. Characteristics include potentially high compile time and size. Loop nests are restructured. Vectorizations that might create false exceptions in rare cases may be performed.

For further information, see [Vectorization Directives on page 76](#).

## 2.11 Inlining Optimization Options

Inlining is the process of replacing a user procedure call with the procedure definition itself. This saves subprogram call overhead and may allow better optimization of the inlined code. If all calls within a loop are inlined, the loop becomes a candidate for parallelization.

The `-h ipan` option specifies automatic inlining. Automatic inlining allows the compiler to automatically select which functions to inline, depending on the inlining level  $n$ . Each  $n$  specifies a different set of heuristics. When `-h ipan` is used alone, the candidates for expansion are all those functions that are present in the input file to the compile step. If `-h ipan` is used in conjunction with `-h ipafrom=source`, the candidates for expansion are those functions present in *source*. For an explanation of each lining level, see [Table 5](#).



The compiler supports the following inlining modes through the indicated options:

- Automatic inlining allows the compiler to automatically select which procedures to inline, depending on the selected inlining level.
- Explicit inlining allows you to explicitly indicate which procedures the compiler should attempt to inline.
- Combined inlining allows you to specify potential targets for inline expansion, while applying the selected level of inlining heuristics.

Cloning is the duplication of a procedure with modifications to the procedure such that it will run more efficiently. The original call site to that procedure is replaced with a call to the duplicate copy.

For example, the compiler will clone a procedure when there are constants in the call site to that procedure. The new clone will replace the associated formal parameter with its constant actual argument.

Automatic cloning is enabled at `-h ipa4` and higher.

The compiler first attempts to inline a call site. If inlining the call site fails, the compiler attempts to clone the procedure for the specific call site.

### 2.11.1 `-h ipan`

Default: `-h ipa3`

The `-h ipan` option allows the compiler to automatically decide which procedures to consider for inlining. Procedures that are potential targets for inline expansion include all the procedures within the input file to the compilation. [Table 5](#) explains what is inlined at each level.

**Table 5. Automatic Inlining Specifications**

Inlining level	Description
0	All inlining is disabled. All inlining compiler directives are ignored.
1	Directive inlining. Inlining is attempted for call sites and routines that are under the control of an inlining pragma directive. See <a href="#">Inlining Directives on page 89</a> for more information about inlining directives.
2	Call nest inlining. Inline a call nest to an arbitrary depth as long as the nest does not exceed some compiler-determined threshold. A call nest can be a leaf routine. The expansion of the call nest must yield straight-line code (code containing no external calls) for any expansion to occur.

Inlining level	Description
3	Constant actual argument inlining. This includes levels 1 and 2, plus any call site that contains a constant actual argument. This is the default inlining level.
4	Tiny routine inlining plus cloning. This includes levels 1, 2, and 3, plus the inlining of very small routines, regardless of where those routines fall in the call graph. The lower limit threshold is an internal compiler parameter. Also, routine cloning is attempted if inlining fails at a given call site.
5	Aggressive interprocedural analysis (IPA). Includes levels 1, 2, 3, and 4. Additionally, Global Constant Propagation is performed. This is the replacement of variables that are statically initialized and never modified anywhere in the user program. The variable is replace with the constant value in its initializer. This applies only to scalar variables.

### 2.11.2 -h ipafrom=*source* [*source*] ...

The `-h ipafrom=source [:source]` option allows you to explicitly indicate the procedures to consider for inline expansion. The *source* arguments identify each file or directory that contains the routines to consider for inlining. Whenever a call is encountered in the input program that matches a routine in *source*, inlining is attempted for that call site.

**Note:** Spaces are not allowed on either side of the equal sign.

All inlining directives are recognized with explicit inlining. For information about inlining directives, see [Inlining Directives on page 89](#).

**Note:** The routines in *source* are not actually loaded with the final program. They are simply templates for the inliner. To have a routine contained in *source* loaded with the program, you must include it in an input file to the compilation.

Use one or more of the following objects in the *source* argument.

**Table 6. File Types**

C or C++ source files	<p>The routines in C or C++ source files are candidates for inline expansion and must contain error-free code.</p> <p>Source files that are acceptable for inlining are files that have one of the following extensions: <code>.C</code>, <code>.C++</code>, <code>.CC</code>, <code>.cxx</code>, <code>.Cxx</code>, <code>.CXX</code>, or <code>.CC</code>.</p>
<i>dir</i>	<p>A directory that contains any of the file types described in this table.</p>

### 2.11.3 Combined Inlining

Combined inlining is invoked by specifying the `-h ipan` and `-h ipafrom=` options on the command line. This inlining mode will look only in *source* for potential targets for expansion, while applying the selected level of inlining heuristics specified by the `-h ipan` option.

## 2.12 Scalar Optimization Options

This section describes scalar optimization options. Each subsection heading shows in parentheses the compiler command with which the option can be used.

### 2.12.1 `-h [no]interchange`

Default: `-h interchange`

The `-h interchange` option allows the compiler to attempt to interchange all loops, a technique that is used to gain performance by having the compiler swap an inner loop with an outer loop. The compiler attempts the interchange only if the interchange will increase performance. Loop interchange is performed only at scalar optimization level 2 or higher.

The `-h nointerchange` option prevents the compiler from attempting to interchange any loops. To disable interchange of loops individually, use the `#pragma _CRI nointerchange` directive.

## 2.12.2 -h scalar*n*

Default:        -h scalar2

The `-h scalarn` option specifies the level of automatic scalar optimization to be performed. Scalar optimization directives are unaffected by this option (see [Scalar Directives on page 84](#)).

The values for *n* are:

- |   |   |
|---|---|
| 0 | Minimal automatic scalar optimization. The <code>-h matherror=errno</code> and the <code>-h zeroinc</code> options are implied by <code>-h scalar0</code> .                                     |
| 1 | Conservative automatic scalar optimization. This level implies <code>-h matherror=abort</code> and <code>-h nozeroinc</code> .  |
| 2 | Aggressive automatic scalar optimization. The scalar optimizations that provide the best application performance are used, with some limitations imposed to allow for faster compilation times. |
| 3 | Very aggressive optimization; compilation times may increase significantly.   |

## 2.12.3 -h [no]zeroinc

Default:        -h nozeroinc

The `-h nozeroinc` option improves run time performance by causing the compiler to assume that constant increment variables (CIVs) in loops are not incremented by expressions with a value of 0.

The `-h zeroinc` option causes the compiler to assume that some constant increment variables (CIVs) in loops might be incremented by 0 for each pass through the loop, preventing generation of optimized code.

For example, in a loop with index *i*, the expression *expr* in the statement *i += expr* can evaluate to 0. This rarely happens in actual code. `-h zeroinc` is the safer and slower option. This option is affected by the `-h scalarn` option (see [-h scalar\*n\* on page 44](#)).

## 2.13 Math Options

This section describes compiler options pertaining to math functions. Each subsection heading shows in parentheses the compiler command with which the option can be used.

### 2.13.1 `-h fpn`

Default: `-h fp2`

The `-h fp` option allows you to control the level of floating-point optimizations. The *n* argument controls the level of allowable optimization; 0 gives the compiler minimum freedom to optimize floating-point operations, while 3 gives it maximum freedom. The higher the level, the lesser the floating-point operations conform to the IEEE standard.

This option is useful for code using algorithms that are unstable but optimizable.

Generally, this is the behavior and usage for each `-h fp` level:

- The `-h fp0` option causes your program's executable code to conform more closely to the IEEE floating-point standard than the default mode (`-h fp2`). When you specify this level, many identity optimizations are disabled, vectorization of floating-point reductions are disabled, executable code is slower than higher floating-point optimization levels, and a scaled complex divide mechanism is enabled that increases the range of complex values that can be handled without producing an underflow.

**Note:** Use the `-h fp0` option only when your code pushes the limits of IEEE accuracy or requires strong IEEE standard conformance.

- The `-h fp1` option performs various, generally safe, non-conforming IEEE optimizations, such as folding `a == a` to `true`, where *a* is a floating point object. At this level, floating-point reassociation<sup>1</sup> is greatly limited, which may affect the performance of your code.

You should never use the `-h fp1` option except when your code pushes the limits of IEEE accuracy or requires strong IEEE standard conformance.

- `-h fp2` — includes optimizations of `-h fp1`.
- `-h fp3` — includes optimizations of `-h fp2`.

You should use the `-h fp3` option when performance is more critical than the level of IEEE standard conformance provided by `-h fp2`.

<sup>1</sup> For example, `a+b+c` is rearranged to `b+a+c`, where *a*, *b*, and *c* are floating point variables.

[Table 7](#) compares the various optimization levels of the `-h fp` option (levels 2 and 3 are usually the same). The table lists some of the optimizations performed; the compiler may perform other optimizations not listed.

**Table 7. Floating-point Optimization Levels**

Optimization Type	fp0	fp1	fp2 (default)	fp3
Complex divisions	Accurate and slower	Accurate and slower	Less accurate (less precision) and faster.	Less accurate (less precision) and faster.
Exponentiation rewrite	None	None	Maximum performance <sup>2</sup>	Maximum performance <sup>2, 3</sup>
Strength reduction	Fast	Fast	Aggressive	Aggressive
Rewrite division as reciprocal equivalent <sup>4</sup>	None	None	Yes	Aggressive
Floating point reductions	Slow	Fast	Fast	Fast
Safety	Maximum	Moderate	Moderate	Low
Expression factoring	None	Yes	Yes	Yes
Expression tree balancing	None	No	Yes	Yes

<sup>2</sup> Rewriting values raised to a constant power into an algebraically equivalent series of multiplications and/or square roots.

<sup>3</sup> Rewriting exponentiations ( $a^b$ ) not previously optimized into the algebraically equivalent form  $\exp(b * \ln(a))$ .

<sup>4</sup> For example,  $x/y$  is transformed to  $x * 1.0/y$ .

### 2.13.2 -h matherror

Default: `-h matherror=abort`

The `-h matherror=method` option specifies the method of error processing used if a standard math function encounters an error. The *method* argument can have one of the following values:

<u>method</u>	<u>Description</u>
abort	If an error is detected, <code>errno</code> is not set. Instead, a message is issued and the program aborts. An exception may be raised.
errno	If an error is detected, <code>errno</code> is set, and the math function returns to the caller. This method is implied by the <code>-h conform</code> , <code>-h scalar0</code> , <code>-O0</code> , <code>-Gn</code> , and <code>-g</code> options.

## 2.14 Debugging Options

This section describes compiler options used for debugging. Each subsection heading shows in parentheses the compiler command with which the option can be used.

### 2.14.1 -G level and -g

The `-G level` and `-g` options enable the generation of debugging information used by symbolic debuggers such as TotalView. These options allow debugging with breakpoints. [Table 8](#) describes the values for the `-G` option.

**Table 8. -G level Definitions**

<i>level</i>	Optimization	Breakpoints allowed on	Debugging	Execution speed
-Gf	Full	Function entry and exit	Limited	Best
-Gp	Partial	Block boundaries	Better	Better
-Gn	None	Every executable statement	Best	Limited

Better debugging information comes at the cost of inhibiting certain optimization techniques, so choose the option that best fits the debugging needs of any particular source file in an application.

The `-g` option is equivalent to `-Gn`. The `-g` option is included for compatibility with earlier versions of the compiler and many other UNIX systems; the `-G` option is the preferred specification. The `-Gn` and `-g` options disable all optimizations and imply `-O0`.

The debugging options take precedence over any conflicting options that appear on the command line. If more than one debugging option appears, the last one specified overrides the others.

Debugging is described in more detail in [Chapter 11, Debugging Cray C and C++ Code on page 125](#).

### 2.14.2 -h [no]bounds (cc)

Default:        -h nobounds

The `-h bounds` option provides checking of pointer and array references to ensure that they are within acceptable boundaries. The `-h nobounds` option disables these checks.

The pointer check verifies that the pointer is greater than 0 and less than the machine memory limit. The array check verifies that the subscript is greater than or equal to 0 and is less than the array size, if declared.

### 2.14.3 -h dir\_check

The `-h dir_check` option enables directive checking at run time. Errors detected at compile time are reported during compilation and so are not reported at run time. The following directives are checked: `shortloop`, `shortloop128`, `collapse`, and the `loop_info` clauses *min\_trips* and *max\_trips*. Violation of a run time check results in an immediate fatal error diagnostic.



**Warning:** Optimization of enclosing and adjacent loops is degraded when run time directive checking is enabled. This capability, though useful for debugging, is not recommended for production runs.

### 2.14.4 -h zero

The `-h zero` option causes stack-allocated memory to be initialized to all zeros.

## 2.15 Compiler Message Options

This section describes compiler options that affect messages. Each subsection heading shows in parentheses the compiler command with which the option can be used.



### 2.15.1 -h msglevel\_*n*

Default:        -h msglevel\_3

The -h msglevel\_*n* option specifies the lowest level of severity of messages to be issued. Messages at the specified level and above are issued. Values for *n* are:

0	Comment
1	Note
2	Caution
3	Warning
4	Error

### 2.15.2 -h [no]message=*n*[:*n*...]

Default:        Determined by -h msglevel\_*n*

The -h [no]message=*n*[:*n*...] option enables or disables specified compiler messages, where *n* is the number of a message to be enabled or disabled. You can specify more than one message number; multiple numbers must be separated by a colon with no intervening spaces. For example, to disable messages CC-174 and CC-9, specify:

```
-h nomessage=174:9
```

The -h [no]message=*n* option overrides -h msglevel\_*n* for the specified messages. If *n* is not a valid message number, it is ignored. Any compiler message except ERROR, INTERNAL, and LIMIT messages can be disabled; attempts to disable these messages by using the -h nomessage=*n* option are ignored.

### 2.15.3 -h report=*args*

The -h report=*args* option generates report messages specified in *args* and lets you direct the specified messages to a file. The *args* field can be any combination of the following options:

f	Writes specified messages to <i>file</i> .v, where <i>file</i> is the source file specified on the command line. If the f option is not specified, messages are written to stderr.
i	Generates inlining optimization messages.
s	Generates scalar optimization messages.
v	Generates vector optimization messages.

No spaces are allowed around the equal sign (=) or any of the *args* codes. For example, the following example prints inlining and scalar optimization messages for `myfile.c`:

```
% cc -h report=is myfile.c
```

The `-h msgs` option also provides optimization messages.

### 2.15.4 `-h [no]abort`

Default: `-h noabort`

The `-h [no]abort` option controls whether a compilation aborts if an error is detected.

### 2.15.5 `-h errorlimit`

Default: `-h errorlimit=100`

The `-h errorlimit[=n]` option specifies the maximum number of error messages the compiler prints before it exits, where *n* is a positive integer. Specifying `-h errorlimit=0` disables exiting on the basis of the number of errors. Specifying `-h errorlimit` with no qualifier is the same as setting *n* to 1.

## 2.16 Compilation Phase Options

This section describes compiler options that affect compilation phases. Each subsection heading shows in parentheses the compiler command with which the option can be used.

### 2.16.1 `-E`

The `-E` option directs the compiler to execute only the preprocessor phase of the compiler. The `-E` and `-P` options are equivalent, except that `-E` directs output to `stdout` and inserts appropriate `#line linenumber` preprocessing directives. The `-E` option takes precedence over the `-h feonly`, `-S`, and `-c` options.

When both the `-E` and `-P` options are specified, the last one specified takes precedence.

### 2.16.2 -P

The `-P` option directs the compiler to execute only the preprocessor phase of the compiler for each source file specified. The preprocessed output for each source file is written to a file with a name that corresponds to the name of the source file and has a `.i` suffix substituted for the suffix of the source file. The `-P` option is similar to the `-E` option, except that `#line linenumber` directives are suppressed, and the preprocessed source does not go to `stdout`. This option takes precedence over `-h feonly`, `-S`, and `-c`.

When both the `-P` and `-E` options are specified, the last one specified takes precedence.

### 2.16.3 -h feonly

The `-h feonly` option limits the compiler to syntax checking. The optimizer and code generator are not executed. This option takes precedence over `-S` and `-c`.

### 2.16.4 -S

The `-S` option compiles the named source files and leaves their assembly language output in the corresponding files suffixed with a `.s`. If this option is used with `-G` or `-g`, debugging information is not generated. This option takes precedence over `-c`.

### 2.16.5 -c

The `-c` option creates a relocatable object file for each named source file but does not link the object files. The relocatable object file name corresponds to the name of the source file. The `.o` suffix is substituted for the suffix of the source file.

### 2.16.6 -#, -##, and -###

The `-#` option produces output indicating each phase of the compilation as it is executed. Each succeeding output line overwrites the previous line.

The `-##` option produces output indicating each phase of the compilation as it is executed.

The `-###` option is the same as `-##`, except the compilation phases are not executed.

### 2.16.7 -w *phase*, "*opt ...*"

The `-w phase` option passes arguments directly to a phase of the compiling system. [Table 9](#) shows the system phases that *phase* can indicate.

**Table 9. -w *phase* Definitions**

<i>phase</i>	System phase	Command
0 (zero)	Compiler	CC and cc
a	Assembler	as
l	Loader	ld

Arguments to be passed to system phases can be entered in either of two styles. If spaces appear within a string to be passed, the string is enclosed in double quotes. When double quotes are not used, spaces cannot appear in the string. Commas can appear wherever spaces normally appear; an option and its argument can be either separated by a comma or not separated. If a comma is part of an argument, it must be preceded by the \ character. For example, any of the following command lines would send -e name and -s to the loader:

```
% cc -Wl, "-e name -s" file.c
```

```
% cc -Wl,-e,name,-s file.c
```

```
% cc -Wl,"-ename",-s file.c
```

Because the preprocessor is built into the compiler, -Wp and -W0 are equivalent.

### 2.16.8 -Y *phase,dirname*

The -Y *phase,dirname* option specifies a new directory (*dirname*) from which the designated *phase* should be executed. The values of *phase* are [Table 10](#).

**Table 10. -Y *phase* Definitions**

<i>phase</i>	System phase	Command
0 (zero)	Compiler	CC, cc
a	Assembler	as
l	Loader	ld

Because there is no separate preprocessor, -Yp and -Y0 are equivalent.

## 2.17 Preprocessing Options

This section describes compiler options that affect preprocessing. Each subsection heading shows in parentheses the compiler command with which the option can be used.

## 2.17.1 -C

The `-C` option retains all comments in the preprocessed source code, except those on preprocessor directive lines. By default, the preprocessor phase strips comments from the source code. This option is useful in combination with the `-P` or `-E` option.

## 2.17.2 -D *macro*[=*def*]

The `-D macro[=def]` option defines *macro* as if it were defined by a `#define` directive. If no *=def* argument is specified, *macro* is defined as 1.

Predefined macros also exist; these are described in [Chapter 9, Using Predefined Macros on page 119](#). Any predefined macro except those required by the standard (see [Macros Required by the C and C++ Standards on page 120](#)) can be redefined by the `-D` option. The `-U` option overrides the `-D` option when the same macro name is specified, regardless of the order of options on the command line.

## 2.17.3 -h [no]pragma=*name*[:*name* ...]

Default: `-h pragma` (no pragmas disabled)

The `[no]pragma=name[:name ...]` option enables or disables the processing of specified directives in the source code, where *name* can be the name of a directive or a word shown in [Table 11](#) to specify a group of directives. More than one name can be specified. Multiple names must be separated by a colon and have no intervening spaces.

**Table 11. -h pragma Directive Processing**

<i>name</i>	Group	Directives affected
all	All	All directives
allinline	Inlining	inline_enable, inline_disable, inline_reset, inline_always, inline_never
allscalar	Scalar optimization	concurrent, nointerchange, noreduction, suppress, unroll/nounroll
allvector	Vectorization	novector, loop_info, hand_tuned, nopattern, novector, permutation, pipeline/nopipeline, prefervector,

<i>name</i>	<b>Group</b>	<b>Directives affected</b>
		safe_address, safe_conditional, shortloop, shortloop128

When using this option to enable or disable individual directives, note that some directives must occur in pairs. For these directives, you must disable both directives if you want to disable either; otherwise, the disabling of one of the directives may cause errors when the other directive is (or is not) present in the compilation unit.

## 2.17.4 **-I *includir***

The `-I includir` option specifies a directory for files named in `#include` directives when the `#include` file names do not have a specified path. Each directory specified must be specified by a separate `-I` option.

The order in which directories are searched for files named on `#include` directives is determined by enclosing the file name in either quotation marks (" ") or angle brackets (< and >).

Directories for `#include "file"` are searched in the following order:

1. Directory of the input file.
2. Directories named in `-I` options, in command-line order.
3. Site-specific and compiler release-specific include files directories.
4. Directory `/usr/include`.

Directories for `#include <file>` are searched in the following order:

1. Directories named in `-I` options, in command-line order.
2. Site-specific and compiler release-specific include files directories.
3. Directory `/usr/include`.

If the `-I` option specifies a directory name that does not begin with a slash (/), the directory is interpreted as relative to the current working directory and not relative to the directory of the input file (if different from the current working directory).

For example:

```
% cc -I. -I yourdir mydir/b.c
```

The preceding command line produces the following search order:

1. `mydir` (`#include "file"` only).
2. Current working directory, specified by `-I`.
3. `yourdir` (relative to the current working directory), specified by `-I yourdir`.
4. Site-specific and compiler release-specific include files directories.
5. Directory `/usr/include`.

### 2.17.5 `-M`

The `-M` option provides information about recompilation dependencies that the source file invokes on `#include` files and other source files. This information is printed in the form expected by `make`. Such dependencies are introduced by the `#include` directive. The output is directed to `stdout`.

### 2.17.6 `-nostdinc`

The `-nostdinc` option stops the preprocessor from searching for include files in the standard directories (`/usr/include` and for Cray C++ also `/usr/include/c++`).

### 2.17.7 `-U`

The `-U` option removes any initial definition of *macro*. Any predefined macro except those required by the standard (see [Macros Required by the C and C++ Standards on page 120](#)) can be undefined by the `-U` option. The `-U` option overrides the `-D` option when the same macro name is specified, regardless of the order of options on the command line.

Predefined macros are described in [Chapter 9, Using Predefined Macros on page 119](#). Macros defined in the system headers are not predefined macros and are not affected by the `-U` option.

## 2.18 Loader Options

This section describes compiler options that affect loader tasks. Each subsection heading shows in parentheses the compiler command with which the option can be used.

### 2.18.1 `-l libname`

The `-l libname` option directs the compiler driver to search for the specified object library file when loading an executable file. To request more than one library file, specify multiple `-l` options.

The compiler driver searches for libraries by prepending `ldir/lib` on the front of `libname` and appending `.a` on the end of it, for each `ldir` that has been specified by using the `-L` option. It uses the first file it finds. See also the `-L` option ([-L \*ldir\* on page 56](#)).

There is no search order dependency for libraries. Default libraries are shown in the following list:

```
libc.a (Cray C++ only)
libu.a
libm.a
libc.a
libsm.a
libf.a
libfi.a
libsci.a
```

If you specify personal libraries by using the `-l` command line option, as in the following example, those libraries are added to the top of the preceding list. (The `-l` option is passed to the loader.)

```
cc -l mylib target.c
```

When the previous command line is issued, the loader looks for a library named `libmylib.a` (following the naming convention) and adds it to the top of the list of default libraries.

### 2.18.2 `-L ldir`

The `-L ldir` option changes the `-l` option search algorithm to look for library files in directory `ldir`. To request more than one library directory, specify multiple `-L` options.

The loader searches for library files in the compiler release-specific directories.

**Note:** Multiple `-L` options are treated cumulatively as if all `ldir` arguments appeared on one `-L` option preceding all `-l` options. Therefore, do not attempt to load functions of the same name from different libraries through the use of alternating `-L` and `-l` options.



### 2.18.3 -o *outfile*

The `-o outfile` option produces an absolute binary file named *outfile*. A file named `a.out` is produced by default. When this option is used in conjunction with the `-c` option and a single source file, a relocatable object file named *outfile* is produced.

## 2.19 Miscellaneous Options

This section describes compiler options that affect general tasks. Each subsection heading shows in parentheses the compiler command with which the option can be used.

### 2.19.1 -h *cpu=target\_system*

The `-h cpu=target_system` option specifies the system on which the absolute binary file is to be executed. The *target\_system* can be `x86-64`, `opteron`, `barcelona`, `shanghai`, or `istanbul`. The default is `x86-64`. You can use the `CRAY_PE_TARGET` environment variable to set the target system; see [Compile Time Environment Variables on page 61](#). For more information, see the *Cray XT Programming Environment User's Guide*.

### 2.19.2 -h *ident=name*

Default:           File name specified on the command line

The `-h ident=name` option changes the *ident* name to *name*. This *name* is used as the module name in the object file (`.o` suffix) and assembler file (`.s` suffix). Regardless of whether the *name* is specified or the default name is used, the following transformations are performed on *name*:

- All `.` characters in the *ident* name are changed to `$`.
- If the *ident* name starts with a number, a `$` is added to the beginning of the *ident* name.

### 2.19.3 -h *keepfiles*

The `-h keepfiles` option prevents the removal of the object (`.o`) files after an executable is created. Normally, the compiler automatically removes these files after linking them to create an executable. Since the original object files are required to instrument a program for performance analysis, if you plan to use CrayPat to conduct performance analysis experiments, you can use this option to preserve the object files.

### 2.19.4 -h network=*nic*

The `-h network=nic` option specifies the target machine's system interconnection network. Currently, the only supported value for *nic* is *seastar*.

### 2.19.5 -h [no]omp

Default:            `-h omp`

The `-h [no]omp` option enables or disables compiler recognition of OpenMP pragmas. For details, see [Chapter 4, Using OpenMP on page 93](#).

### 2.19.6 -h prototype\_intrinsics

The `-h prototype_intrinsics` option simulates the effect of including `intrinsics.h` at the beginning of a compilation. Use this option if the source code does not include the `intrinsics.h` statement and you cannot modify the code. This option is off by default. For details, see [Appendix D, Using Intrinsic Functions on page 171](#).

### 2.19.7 -h task*n*

Default:            `-h task0`

The `-h taskn` option enables tasking in applications that contain OpenMP directives.

<i>n</i>	Description
0	Disables tasking. OpenMP directives are ignored. Using this option can reduce compile time and the size of the executable. The <code>-h task0</code> option is compatible with all vectorization and scalar optimization levels.
1	The <code>-h task1</code> option specifies user tasking, so OpenMP directives are recognized. No level for scalar optimization is enabled automatically. The <code>-h task1</code> option is compatible with all vectorization and scalar optimization levels.

## 2.19.8 -h [no]threadsafe

Default:        -h threadsafe

The `-h [no]threadsafe` option enables or disables the generation of threadsafe code. Code that is threadsafe can be used with pthreads and OpenMP. This option is not binary-compatible with code generated by Cray C 8.1 or Cray C++ 5.1 and earlier compilers. Users who need binary compatibility with previously compiled code can use `-h nothreadsafe`, which causes the compiler to be compatible with Cray C 8.1 or Cray C++ 5.1 and earlier compilers at the expense of not being threadsafe.

C or C++ code compiled with `-h threadsafe` (the default) cannot be linked with C or C++ code compiled with `-h nothreadsafe` or with code compiled with a Cray C 8.1, Cray C++ 5.1, or earlier compiler.

## 2.19.9 -h upc (cc)

The `-h upc` option enables compilation of Unified Parallel C (UPC) code. UPC is a C language extension for parallel program development that allows you to explicitly specify parallel programming through language syntax rather than through library functions such as are used in MPI or SHMEM.

The Cray implementation of UPC is discussed in [Chapter 5, Using Cray Unified Parallel C \(UPC\) on page 99](#).

## 2.19.10 -v

The `-v` option displays compiler version information. If the command line specifies no source file, no compilation occurs.

Version information consists of the product name, the version number, and the current date and time, as shown in the following example:

```
% CC -v
/opt/cray/xt-asyncpe/2.5/bin/CC: INFO: native target is being used
Cray C++ : Version 7.1.0.129 Thu May 21, 2009 12:59:44
```

## 2.19.11 -x npes

The `-x npes` option specifies the number of processing elements (PEs) that will be specified through aprun at job launch. The value for *npes* ranges from 1 through 65535 inclusive.

Ensure that you compile all object files with the same `-x npes` value and run the resulting executable with that number of PEs. If you use mixed `-x npes` values or if the number of PEs provided at run time differs from the `-x npes` value, program behavior is undefined.

The number of PEs to use cannot be changed at load or run time. You must recompile the program with a different value for *npes* to change the number of PEs.

For further information about running applications, see the *Cray XT Programming Environment User's Guide* or the `aprun(1)` man page.

## 2.20 Command Line Examples

The following examples illustrate a variety of command lines for the C and C++ compiler commands:

**Example 1.** `CC -x8 -h instantiate=all myprog.C`

This example compiles `myprog.C`, fixes the number of processing elements to 8, and instantiates all template entities declared or referenced in the compilation unit.

```
% CC -x8 -h instantiate=all myprog.C
```

**Example 2.** `CC -h conform -h noautoinstantiate myprog.C`

This example compiles `myprog.C`. The `-h conform` option specifies strict conformance to the ISO C++ standard. No automatic instantiation of templates is performed.

```
% CC -h conform -h noautoinstantiate myprog.C
```

**Example 3.** `cc -c -h ipal myprog.c subprog.c`

This example compiles input files `myprog.c` and `subprog.c`. The `-c` option tells the compiler to create object files `myprog.o` and `subprog.o` but not call the loader. Option `-h ipal` tells the compiler to inline function calls marked with the `inline_always` pragma.

```
% cc -c -h ipal myprog.c subprog.c
```

**Example 4.** `cc -I. disc.c vend.c`

This example specifies that the compiler search the current working directory, represented by a period (`.`), for `#include` files before searching the default `#include` file locations.

```
% cc -I. disc.c vend.c
```

**Example 5.** `cc -P -D DEBUG newprog.c`

This example specifies that source file `newprog.c` be preprocessed only. Compilation and linking are suppressed. In addition, the macro `DEBUG` is defined.

```
% cc -P -D DEBUG newprog.c
```

**Example 6.** `cc -c -h report=s mydata1.c`

This example compiles `mydata1.c`, creates object file `mydata1.o`, and produces a scalar optimization report to `stdout`.

```
% cc -c -h report=s mydata1.c
```

**Example 7.** `CC -h ipa5,report=if myfile.C`

This example compiles `myfile.C` and tells the compiler to attempt to aggressively inline calls to functions defined within `myfile.C`. An inlining report is directed to `myfile.V`.

```
% CC -h ipa5,report=if myfile.C
```

## 2.21 Compile Time Environment Variables

The following environment variables are used during compilation.

<u>Variable</u>	<u>Description</u>
-----------------	--------------------

CRAYOLDCPPLIB	
---------------	--

When set to a nonzero value, enables C++ code to use the following nonstandard Cray C++ headers files:

- `common.h`
- `complex.h`
- `fstream.h`
- `generic.h`
- `iomanip.h`
- `iostream.h`
- `stdiostream.h`
- `stream.h`
- `strstream.h`
- `vector.h`

If you want to use the standard header files, your code may require modification to compile successfully. For more information, see [Appendix B, Using Cray C and C++ Dialects on page 151](#).

**Note:** Setting the `CRAYOLDCPPLIB` environment variable disables exception handling, unless you compile with the `-h exceptions` option.

**CRAY\_PE\_TARGET**

Specifies the target system to be used as the default in all compilations. Valid options are `x86-64`, `opteron`, `barcelona`, `shanghai`, and `istanbul`. It is overridden by the `-hcpu=target` option (see [-h cpu=\*target\\_system\* on page 57](#)). If unset, the default target is `x86-64`.

**CRI\_CC\_OPTIONS****CRI\_cc\_OPTIONS**

Specifies command line options that are applied to all compilations. Options specified by this environment variable are added following the options specified directly on the command line. This is especially useful for adding options to compilations done with build tools.

Identifies your requirements for native language, local customs, and coded character set with regard to compiler messages.

Controls the format in which you receive compiler messages.

Specifies the message system catalogs that should be used.

Specifies the number of processes used for simultaneous compilations. The default is 1. When more than one source file is specified on the command line, compilations may be multiprocessed by setting the environment variable `NPROC` to a value greater than 1. You can set `NPROC` to any value; however, large values can overload the system.

## 2.22 Run Time Environment Variables

### APRUN\_XFER\_LIMITS

The default behavior for forwarding of user resource limits to compute nodes has changed in CLE 2.2. The `aprun` utility no longer forwards its user resource limits to each compute node (except for `RLIMIT_CORE` and `RLIMIT_CPU`, which are always forwarded). This CLE 2.2 default is the recommended behavior for CCE users.

In CLE 2.1, it is recommended that you set the `APRUN_XFER_LIMITS` environment variable to 0 (`export APRUN_XFER_LIMITS=0` or `setenv APRUN_XFER_LIMITS 0`) to disable the forwarding of user resource limits. For more information, see the `getrlimit(P)` man page.

**Note:** It is recommended that CLE 2.1 site administrators set `APRUN_XFER_LIMITS` to 0 in the default shells (`/etc/bash.bashrc.local`, for example) to avoid run time segmentation faults due to exceeding the stack size limit.

## 2.23 OpenMP Environment Variables

For Cray-specific information about OpenMP environment variables, see [Chapter 4, Using OpenMP on page 93](#). For documentation of standard OpenMP environment variables, see the *OpenMP Application Program Interface Version 3.0 May 2008* standard (<http://openmp.org/wp/openmp-specifications/>).





# Using #pragma Directives [3]

---

The #pragma directives are used within the source program to request certain kinds of special processing. The directives are part of the C and C++ languages, but the meaning of any #pragma directive is defined by the implementation. #pragma directives are expressed in the following form:

```
#pragma [_CRI] identifier [arguments]
```

The <sub>\_CRI</sub> specification is optional; it ensures that the compiler will issue a message concerning any directives that it does not recognize. Diagnostics are not generated for directives that do not contain the <sub>\_CRI</sub> specification.

These directives are classified according to the following types:

- General ([General Directives on page 67](#))
- Instantiation, Cray C++ ([Instantiation Directives on page 76](#))
- Vectorization ([Vectorization Directives on page 76](#))
- Scalar ([Scalar Directives on page 84](#))
- Inlining ([Inlining Directives on page 89](#))

Macro expansion occurs on the directive line after the directive name. That is, macro expansion is applied only to *arguments*.

**Note:** OpenMP #pragma directives are described in [Chapter 4, Using OpenMP on page 93](#).

At the beginning of each section that describes a directive, information is included about the compilers that allow the use of the directive and the scope of the directive. Unless otherwise noted, the following default information applies to each directive:

Compiler:      Cray C and Cray C++

Scope:          Local and global

The scoping list may also indicate that a directive has a lexical block scope. A lexical block is the scope within which a directive is on or off and is bounded by the opening curly brace just before the directive was declared and the corresponding closing curly brace. Only applicable executable statements within the lexical block are affected as indicated by the directive. The lexical block does not include the statements contained within a procedure that is called from the lexical block.

This example code fragment shows the lexical block for the `upc strict` and `upc relaxed` directives:

```
void Example(void)
{
    #pragma _CRI upc strict // UPC strict state is on
    ...

    {
        ... // UPC strict state is still on
        #pragma _CRI upc relaxed // UPC strict state is now off
        ...
    }

    // UPC strict state is back on
    ...
}
```

## 3.1 Protecting Directives

To ensure that your directives are interpreted only by the Cray C and C++ compilers, use the following coding technique in which *directive* is the name of the directive:

```
#if _CRAYC
    #pragma _CRI directive
#endif
```

This ensures that other compilers used to compile this code will not interpret the directive. Some compilers diagnose any directives that they do not recognize. The Cray C and C++ compilers diagnose directives that are not recognized only if the `_CRI` specification is used.

## 3.2 Directives in Cray C++

C++ prohibits referencing undeclared objects or functions. Objects and functions must be declared prior to using them in a `#pragma` directive. This is not always the case with C.

Some `#pragma` directives take function names as arguments (for example: `#pragma _CRI weak`, `#pragma _CRI suppress`, and `#pragma _CRI inline_always name [ ,name ... ]`). Member functions and qualified names are allowed for these directives.

## 3.3 Loop Directives

Many directives apply to groups. Unless otherwise noted, these directives must appear before a `for`, `while`, or `do while` loop. These directives may also appear before a label for `if...goto` loops. If a loop directive appears before a label that is not the top of an `if...goto` loop, it is ignored.

### 3.4 Alternative Directive Form: `_Pragma`

Compiler directives can also be specified in the following form, which has the advantage in that it can appear inside macro definitions:

```
_Pragma( "_CRI identifier" );
```

This form has the same effect as using the `#pragma` form, except that everything that appeared on the line following the `#pragma` must now appear inside the double quotation marks and parentheses. The expression inside the parentheses must be a single string literal; it cannot be a macro that expands into a string literal. `_Pragma` is an extension to the C and C++ standards.

The following is an example using the `#pragma` form:

```
#pragma _CRI concurrent
```

The following is the same example using the alternative form:

```
_Pragma( "_CRI concurrent" );
```

In the following example, the loop automatically vectorizes wherever the macro is used:

```
#define _str( _X ) # _X
#define COPY( _A, _B, _N

{
    int i;
    _Pragma( "_CRI concurrent" )
    _Pragma( _str( _CRI loop_info cache_nt( _B ) ) )
    for ( i = 0; i < _N; i++ ) {
        _A[i] = _B[i];
    }
}

void
copy_data( int *a, int *b, int n )
{
    COPY( a, b, n );
}
```

Macros are expanded in the string literal argument for `_Pragma` in an identical fashion to the general specification of a `#pragma` directive.

### 3.5 General Directives

General directives specify compiler actions that are specific to the directive and have no similarities to the other types of directives. The following sections describe general directives.

### 3.5.1 [no]bounds Directive

The `bounds` directive specifies that pointer and array references are to be checked. The `nobounds` directive specifies that this checking is to be disabled.

When bounds checking is in effect, pointer references are checked to ensure that they are neither 0 nor greater than the machine memory limit. Array references are checked to ensure that the array subscript is not less than 0 or greater than or equal to the declared size of the array.

Both directives may be used only within function bodies. They apply until the end of the function body or until another `bounds/nobounds` directive appears. They ignore block boundaries.

These directives have the following format:

```
#pragma _CRI bounds
#pragma _CRI nobounds
```

The following example illustrates the use of the `bounds` directive:

```
int a[30];
#pragma _CRI bounds
void f(void)
{
    int x;
    x = a[30];
    .
    .
    .
}
```

### 3.5.2 duplicate Directive

Scope:           Global

The `duplicate` directive lets you provide additional, externally visible names for specified functions. You can specify duplicate names for functions by using a directive with one of the following forms:

```
#pragma _CRI duplicate actual as dupname...
#pragma _CRI duplicate actual as (dupname...)
```

The *actual* argument is the name of the actual function to which duplicate names will be assigned. The *dupname* list contains the duplicate names that will be assigned to the actual function. The *dupname* list may be optionally parenthesized. The word *as* must appear as shown between the *actual* argument and the comma-separated list of *dupname* arguments.

The `duplicate` directive can appear anywhere in the source file and it must appear in global scope. The actual name specified on the directive line must be defined somewhere in the source as an externally accessible function; the actual function cannot have a static storage class.

The following example illustrates the use of the duplicate directive:

```
#include <complex.h>

extern void maxhits(void);

#pragma _CRI duplicate maxhits as count, quantity      /* OK */

void maxhits(void)
{
    #pragma _CRI duplicate maxhits as tempcount
    /* Error: #pragma _CRI duplicate can't appear in local scope */
}

double _Complex minhits;

#pragma _CRI duplicate minhits as lower_limit
/* Error: minhits is not declared as a function */

extern void derivspeed(void);

#pragma _CRI duplicate derivspeed as accel
/* Error: derivspeed is not defined */

static void endtime(void)
{
}

#pragma _CRI duplicate endtime as limit
/* Error: endtime is defined as a static function */
```

Because duplicate names are simply additional names for functions and are not functions themselves, they cannot be declared or defined anywhere in the compilation unit. To avoid aliasing problems, duplicate names may not be referenced anywhere within the source file, including appearances on other directives. In other words, duplicate names may only be referenced from outside the compilation unit in which they are defined.

The following example references duplicate names:

```
void converter(void)
{
    structured(void);
}

#pragma _CRI duplicate converter as factor, multiplier /* OK */

void remainder(void)
{
}

#pragma _CRI duplicate remainder as factor, structured
/* Error: factor and structured are referenced in this file */
```

Duplicate names can be used to provide alternate external names for functions, as shown in the following examples.

**main.c:**

```
extern void fctn(void), FCTN(void);

main()
{
    fctn();
    FCTN();
}
```

**fctn.c:**

```
#include <stdio.h>

void fctn(void)
{
    printf("Hello world\n");
}

#pragma _CRI duplicate fctn as FCTN
```

Files `main.c` and `fctn.c` are compiled and linked using the following command line:

```
% cc main.c fctn.c
```

When the executable file `a.out` is run, the program generates the following output:

```
Hello world
Hello world
```

### 3.5.3 message Directive

The message directive directs the compiler to write the message defined by *text* to `stderr` as a warning message. Unlike the error directive, the compiler continues after processing a message directive. The format of this directive is as follows:

```
#pragma _CRI message "text"
```

The following example illustrates the use of the message compiler directive:

```
#define FLAG 1

#ifdef FLAG
#pragma _CRI message "FLAG is Set"
#else
#pragma _CRI message "FLAG is NOT Set"
#endif
```

### 3.5.4 `cache` Directive

The `cache` directive asserts that all memory operations with the specified symbols as the base are to be allocated in cache. This is an advisory directive. The `cache` directive is meaningful for stores in that it allows the user to override a decision made by the automatic cache management. This directive may be locally overridden by the use of a `#pragma loop_info` directive. This directive overrides automatic cache management decisions (see `-h cachem`).

To use the directive, you must place it only in the specification part, before any executable statement.

The format of the `cache` directive is:

```
#pragma _CRI cache base_name [ ,base_name ... ]
```

*base\_name*      The base name of the object that should be placed into the cache. This can be the base name of any object such as an array, scalar structure, and so on, without member references like `C[10]`. If you specify a pointer in the list, only the references, not the pointer itself, are cached.

### 3.5.5 `cache_nt` Directive

The `cache_nt` directive is an advisory directive that specifies objects that should use non-temporal reads and writes. Use this directive to identify objects that should not be placed in cache.

The format of the `cache_nt` directive is:

```
#pragma _CRI cache_nt base_name [ ,base_name ... ]
```

*base\_name*      The base name of the object that should use non-temporal reads and writes. This can be the base name of any object such as an array, scalar structure, and so on, without member references like `C[10]`. If you specify a pointer in the list, only the references, not the pointer itself, have the cache non-temporal property.

This directive overrides the automatic cache management level that was specified using the `-h cachem` option on the compiler command line. This directive may be overridden locally by use of a `loop_info` directive.

### 3.5.6 `ident` Directive

The `ident` pragma directs the compiler to store the string indicated by *text* into the object (`.o`) file. This can be used to place a source identification string into an object file.

The format of this directive is as follows:

```
#pragma _CRI ident text
```

### 3.5.7 `[no]opt` Directive

Scope:           Global

The `noopt` directive disables all automatic optimizations and causes optimization directives to be ignored in the source code that follows the directive. Disabling optimization removes various sources of potential confusion in debugging. The `opt` directive restores the state specified on the command line for automatic optimization and directive recognition. These directives have global scope and override related command line options.

The format of these directives is as follows:

```
#pragma _CRI opt  
#pragma _CRI noopt
```

The following example illustrates the use of the `opt` and `noopt` compiler directives:

```
#include <stdio.h>  
  
void sub1(void)  
{  
    printf("In sub1, default optimization\n");  
}  
  
#pragma _CRI noopt  
void sub2(void)  
{  
    printf("In sub2, optimization disabled\n");  
}  
#pragma _CRI opt  
  
void sub3(void)  
{  
    printf("In sub3, optimization enabled\n");  
}  
  
main()  
{  
    printf("Start main\n");  
    sub1();  
    sub2();  
    sub3();  
}
```



### 3.5.8 autothread, noautothread Directives

Scope:           Local

The `autothread` and `noautothread` directives turn autothreading on and off for selected blocks of code.

The format of these directives is as follows:

```
#pragma _CRI autothread
#pragma _CRI noautothread
```

### 3.5.9 Probability Directives

The `probability`, `probability_almost_always`, and `probability_almost_never` directives specify information used by interprocedural analysis (IPA) and the optimizer to produce faster code sequences. The specified probability is a hint, rather than a statement of fact. You can also specify `almost_never` and `almost_always` by using the values 0.0 and 1.0, respectively.

These directives have the following format:

```
#pragma probability const
#pragma probability_almost_always
#pragma probability_almost_never
```

*const* is an expression that evaluates to a floating point constant at compilation time. (0.0 <= *const* <= 1.0.)

These directives can appear anywhere executable code is legal.

Each directive applies to the block of code where it appears. It is important to realize that the directive should not be applied to a conditional test directly; rather, it should be used to indicate the relative probability of a `then` or `else` branch being executed.

Example:

```
    if ( a[i] > b[i] ) {
#pragma probability 0.3
        a[i] = b[i];
    }
```

This example states that the probability of entering the block of code with the assignment statement is 0.3 or 30%. This also means that `a[i]` is expected to be greater than `b[i]` 30% of the time.

Note that the `probability` directive appears within the conditional block of code, rather than before it. This removes some of the ambiguity that has plagued other implementations that tie the directive directly to the conditional code.

This information is used to guide inlining decisions, branch elimination optimizations, branch hint marking, and the choice of the optimal algorithmic approach to the vectorization of conditional code.

The following GCC-style intrinsic is also accepted when it appears in a conditional test:

```
__builtin_expect( expr, const )
```

The following example:

```
if ( __builtin_expect( a[i] > b[i], 0 ) ) {  
    a[i] = b[i];  
}
```

is roughly equivalent to:

```
if ( a[i] > b[i] ) {  
#pragma _CRI probability_almost_never  
    a[i] = b[i];  
}
```

### 3.5.10 weak Directive

Scope:           Global

The weak directive specifies an external identifier that may remain unresolved throughout the compilation. A weak external reference can be a reference to a function or to a data object. A weak external does not increase the total memory requirements of your program.

Declaring an object as a weak external directs the loader to do one of these tasks:

- Link the object only if it is already linked (that is, if a strong reference exists); otherwise, leave it as an unsatisfied external. The loader does not display an unsatisfied external message if weak references are not resolved.
- If a strong reference is specified in the weak directive, resolve all weak references to it.

**Note:** The loader treats weak externals as unsatisfied externals, so they remain silently unresolved if no strong reference occurs during compilation. Thus, it is your responsibility to ensure that run time references to weak external names do not occur unless the loader (using some "strong" reference elsewhere) has actually loaded the entry point in question.

These are the forms of the weak directive:

```
#pragma _CRI weak var  
#pragma _CRI weak sym1 = sym2
```

*var*                   The name of an external

*sym1*                 Defines an externally visible weak symbol

*sym2*                 Defines an externally visible strong symbol defined in the current compilation.

The first form allows you to declare one or more weak references on one line. The second form allows you to assign a strong reference to a weak reference.

The weak directive must appear at global scope.

The attributes that weak externals must have depend on the form of the weak directive that you use:

- First form, weak externals must be declared, but not defined or initialized, in the source file.
- Second form, weak externals may be declared, but not defined or initialized, in the source file.
- Either form, weak externals cannot be declared with a `static` storage class.

The following example illustrates these restrictions:

```
extern long x;
#pragma _CRI weak x /* x is a weak external data object */
extern void f(void);
#pragma _CRI weak f /* f is a weak external function */

extern void g(void);
#pragma _CRI weak g=fun; /* g is a weak external function
                        with a strong reference to fun */

long y = 4;
#pragma _CRI weak y /* ERROR - y is actually defined */

static long z;
#pragma _CRI weak z /* ERROR - z is declared static */

void fctn(void)
{
#pragma _CRI weak a /* ERROR - directive must be at global scope */
}
```

## 3.6 Instantiation Directives

The Cray C++ compiler recognizes three instantiation directives. Instantiation directives can be used to control the instantiation of specific template entities or sets of template entities. The following directives are described in detail in [Instantiation #pragma Directives on page 111](#):

- `#pragma _CRI instantiate`
- `#pragma _CRI do_not_instantiate`
- `#pragma _CRI can_instantiate`
- The `#pragma _CRI instantiate` directive causes a specified entity to be instantiated.
- The `#pragma _CRI do_not_instantiate` directive suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition is supplied.
- The `#pragma _CRI can_instantiate` directive indicates that a specified entity can be instantiated in the current compilation but need not be. It is used in conjunction with automatic instantiation to indicate potential sites for instantiation if the template entity is deemed to be required by the compiler.

For more information about template instantiation, see [Chapter 7, Using Cray C++ Template Instantiation on page 105](#).

## 3.7 Vectorization Directives

Because vector operations cannot be expressed directly in Cray C and C++, the compilers must be capable of vectorization, which means transforming scalar operations into equivalent vector operations. The candidates for vectorization are operations in loops and assignments of structures.

The subsections that follow describe the compiler directives used to control vectorization.

### 3.7.1 `hand_tuned` Directive

The format of this directive is:

```
#pragma _CRI hand_tuned
```

This directive asserts that the code in the loop nest has been arranged by hand for maximum performance, and the compiler should restrict some of the more aggressive automatic expression rewrites. The compiler should still fully optimize and vectorize the loop within the constraints of the directive.

The `hand_tuned` directive applies to the next loop in the same manner as the `concurrent` and `safe_address` directives.



**Warning:** Use of this directive may severely impede performance. Use carefully and evaluate before and after performance.

### 3.7.2 `loop_info` Directive

Scope:            Local

The `loop_info` directive allows additional information to be specified about the behavior of a loop, including run time trip count and hints on cache allocation strategy.

In regard to trip count information, the `loop_info` directive is similar to the `shortloop` or `shortloop128` directive but provides more information to the optimizer and can produce faster code sequences. `loop_info` is used immediately before a `for` loop to indicate minimum, maximum, or estimated trip count. The compiler will diagnose misuse at compile time (when able) or when option `-h dir_check` is specified at run time.

For cache allocation hints, the `loop_info` directive can be used to override default settings, `cache` or `cache_nt` directives, or override automatic cache management decisions. The cache hints are local and apply only to the specified loop nest.

The format of this directive is:

```
#pragma _CRI loop_info [min_trips(c)] [est_trips(c)] [max_trips(c)]
[cache( symbol[,symbol ...] )]
[cache_nt(symbol[,symbol ...] ) ]
[prefetch] [noprefetch]
```

The `prefetch` and `noprefetch` options are deferred.

*c*                    An expression that evaluates to an integer constant at compilation time.

*min\_trips*        Specifies guaranteed minimum number of trips.

*est\_trips*        Specifies estimated or average number of trips.

*max\_trips*        Specifies guaranteed maximum number of trips.

*cache*            Specifies that *symbol* is to be allocated in cache; this is the default if no hint is specified and the `cache_nt` directive is not specified.

*cache\_nt*        Specifies that *symbol* is to use non-temporal reads and writes.

*prefetch*        Specifies a preference that prefetches be performed for the following loop.

*noprefetch*      Specifies a preference that no prefetches be performed for the following loop.

*symbol*            The base name of the object that should not be placed into the cache. This can be the base name of any object (such as an array or scalar structure) without member references like `C[10]`. If you specify a pointer in the list, only the references, not the pointer itself, have the no cache allocate property.

#### **Example 8. Trip counts**

In the following example, the minimum trip count is 1 and the maximum trip count is 1000:

```
void
loop_info( double *restrict a, double *restrict b, double s1, int n )
{
    int i;

    #pragma _CRI loop_info min_trips(1) max_trips(1000), cache_nt(b)
    for (i = 0; i < n; i++) {
        if(a[i] != 0.0) {
            a[i] = a[i] + b[i]*s1;
        }
    }
}
```

### **3.7.3 loop\_info prefer\_thread, prefer\_nothread Directives**

Scope:            Local

Use these directives to indicate a preference for turning threading on or off for selected loops. Use the `loop_info prefer_thread` directive to indicate your preference that the loop following the directive be threaded. The `loop_info prefer_nothread` indicates your preference that the loop following the directive should not be threaded.

The format of these directives is:

```
#pragma _CRI loop_info prefer_thread
#pragma _CRI loop_info prefer_nothread
```

### **3.7.4 nopattern Directive**

Scope:            Local

The `nopattern` directive disables pattern matching for the loop immediately following the directive.

The format of this directive is as follows:

```
#pragma _CRI nopattern
```

By default, the compiler detects coding patterns in source code sequences and replaces these sequences with calls to optimized library functions. In most cases, this replacement improves performance. There are cases, however, in which this substitution degrades performance. This can occur, for example, in loops with very low trip counts. In such a case, you can use the `nopattern` directive to disable pattern matching and cause the compiler to generate inline code.

In the following example, placing the `nopattern` directive in front of the outer loop of a nested loop turns off pattern matching for the matrix multiply that takes place inside the inner loop:

```
double a[100][100], b[100][100], c[100][100];

void nopat(int n)
{
    int i, j, k;

    #pragma _CRI nopattern
    for (i=0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            for (k = 0; k < n; ++k) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

### 3.7.5 novector Directive

Scope:           Local

The `novector` directive directs the compiler to not vectorize the loop immediately following the directive. It overrides any other vectorization-related directives, as well as the `-h vector` command line option. The format of this directive is as follows:

```
#pragma _CRI novector
```

The following example illustrates the use of the `novector` compiler directive:

```
#pragma _CRI novector
for (i = 0; i < h; i++) {     /* Loop not vectorized */
    a[i] = b[i] + c[i];
}
```

### 3.7.6 permutation Directive

The `permutation` directive specifies that an integer array has no repeated values. This directive is useful when the integer array is used as a subscript for another array (vector-valued subscript). This directive may improve code performance.

This directive has the following format:

```
#pragma _CRI permutation symbol [, symbol ] ...
```

In a sequence of array accesses that read array element values from the specified symbols with no intervening accesses that modify the array element values, each of the accessed elements will have a distinct value.

When an array with a vector-valued subscript appears on the left side of the equal sign in a loop, many-to-one assignment is possible. Many-to-one assignment occurs if any repeated elements exist in the subscripting array. If it is known that the integer array is used merely to permute the elements of the subscripted array, it can often be determined that many-to-one assignment does not exist with that array reference.

Sometimes a vector-valued subscript is used as a means of indirect addressing because the elements of interest in an array are sparsely distributed; in this case, an integer array is used to select only the desired elements, and no repeated elements exist in the integer array, as in the following example:

```
int *ipnt;
#pragma permutation ipnt
...
for ( i = 0; i < N; i++ ) {
    a[ipnt[i]] = b[i] + c[i];
}
```

The permutation directive does not apply to the array `a`. Rather, it applies to the pointer used to index into it, `ipnt`. By knowing that `ipnt` is a permutation, the compiler can safely generate an unordered scatter for the write to `a`.

### 3.7.7 [no]pipeline Directive

Software-based vector pipelining (software vector pipelining) provides additional optimization beyond the normal hardware-based vector pipelining. In software vector pipelining, the compiler analyzes all vector loops and automatically attempts to pipeline a loop if doing so can be expected to produce a significant performance gain. This optimization also performs any necessary loop unrolling.

In some cases the compiler either does not pipeline a loop that could be pipelined or pipelines a loop without producing performance gains. In these situations, you can use the `pipeline` or `nopipeline` directive to advise the compiler to pipeline or not pipeline the loop immediately following the directive.

Software vector pipelining is valid only for the innermost loop of a loop nest.

The `pipeline` and `nopipeline` directives are advisory only. While you can use the `nopipeline` directive to inhibit automatic pipelining, and you can use the `pipeline` directive to attempt to override the compiler's decision not to pipeline a loop, you cannot force the compiler to pipeline a loop that cannot be pipelined.

Loops that have been pipelined are so noted in loopmark listing messages.

The formats of the pipelining directives are as follows:

```
#pragma _CRI pipeline
#pragma _CRI nopipeline
```



### 3.7.8 `prefervector` Directive

Scope:           Local

The `prefervector` pragma directs the compiler to vectorize the loop immediately following the directive if the loop contains more than one loop in the nest that can be vectorized. The directive states a vectorization preference and does not guarantee that the loop has no memory-dependence hazard.

The format of this directive is:

```
#pragma _CRI prefervector
```

The following example illustrates the use of the `prefervector` directive:

```
float a[1000], b[100][1000];

void
f(int m, int n)
{
    int i, j;

    #pragma _CRI prefervector
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            a[i] += b[j][i];
        }
    }
}
```

In this example, both loops can be vectorized, but the directive directs the compiler to vectorize the outer `for` loop. Without the directive and without any knowledge of `n` and `m`, the compiler would vectorize the inner loop.

### 3.7.9 `pgo loop_info` Directive

Scope:           Local

The format of this directive is as follows:

```
#pragma _CRI pgo loop_info
```

The `pgo loop_info` directive enables profile-guided optimizations by tagging loopmark information as having come from profiling. For information about CrayPat and profile information, see the *Using Cray Performance Analysis Tools* guide.

### 3.7.10 `safe_address` Directive

Scope:           Local

The format of this directive is as follows:

```
#pragma _CRI safe_address
```

The `safe_address` directive specifies that it is safe to speculatively execute memory references within all conditional branches of a loop. In other words, you know that these memory references can be safely executed in each iteration of the loop.

For most code, the `safe_address` directive can improve performance significantly by preloading vector expressions. However, most loops do not require this directive to have preloading performed. The directive is required only when the safety of the operation cannot be determined or index expressions are very complicated.

The `safe_address` directive is an advisory directive. That is, the compiler may override the directive if it determines the directive is not beneficial.

If you do not use the directive on a loop and the compiler determines that it would benefit from the directive, it issues a message indicating such. The message is similar to this:

```
CC-6375 cc: VECTOR File = ctest.c, Line = 6
      A loop would benefit from "#pragma safe_address".
```

If you use the directive on a loop and the compiler determines that it does not benefit from the directive, it issues a message that states the directive is superfluous and can be removed.

To see the messages, you must use the `-h report=v` or `-h msgs` option.



**Caution:** Incorrect use of the directive can result in segmentation faults, bus errors, or excessive page faulting. However, it should not result in incorrect answers. Incorrect usage can result in very severe performance degradations or program aborts.

In the example below, the compiler will not preload vector expressions, because the value of `j` is unknown. However, if you know that references to `b[i][j]` is safe to evaluate for all iterations of the loop, regardless of the condition, you can use the `safe_address` directive for this loop as shown below:

```
void x3( double a[restrict 1000], int j )
{
    int i;
    #pragma _CRI safe_address
    for ( i = 0; i < 1000; i++ ) {
        if ( a[i] != 0.0 ) {
            b[j][i] = 0.0;
        }
    }
}
```

With the directive, the compiler can safely load `b[i][j]` as a vector, merge `0.0` where the condition is true, and store the resulting vector safely.

### 3.7.11 safe\_conditional Directive

The `safe_conditional` directive specifies that it is safe to execute all references and operations within all conditional branches of a loop. In other words, you know that these memory references can be safely executed in each iteration of the loop. This directive specifies that memory and arithmetic operations are safe.

This directive applies to scalar and vector loop nests. It can improve performance by allowing the hoisting of invariant expressions from conditional code and by allowing prefetching of memory references.

The `safe_conditional` directive is an advisory directive. That is, the compiler may override the directive if it determines the directive is not beneficial.



**Caution:** Incorrect use of the directive can result in segmentation faults, bus errors, excessive page faulting, or arithmetic aborts. However, it should not result in incorrect answers. Incorrect usage can result in severe performance degradations or program aborts.

The `safe_conditional` directive has the following format:

```
#pragma _CRI safe_conditional
```

In the following example, without the `safe_conditional` directive, the compiler cannot precompute the invariant expression `s1*s2` because their values are unknown and may cause an arithmetic trap if executed unconditionally. However, if you know that the condition is true at least once, then `s1*s2` is safe to speculatively execute. The `safe_conditional` compiler directive can be used to imply the safety of the operation. With the directive, the compiler evaluates `s1*s2` outside of the loop, rather than under control of the conditional code. In addition, all control flow is removed from the body of the vector loop, because `s1*s2` no longer poses a safety risk.

```
void
safe_cond( double a[restrict 1000], double s1, double s2 )
{
    int i;

    #pragma _CRI safe_conditional
    for (i = 0; i < 1000; i++) {
        if( a[i] != 0.0 ) {
            a[i] = a[i] + s1*s2;
        }
    }
}
```

### 3.7.12 `shortloop` and `shortloop128` Directives

Scope:            Local

The `shortloop` compiler directive identifies loops that execute with a maximum iteration count of 64 and a minimum iteration count of 1. The `shortloop128` compiler directive identifies loops that execute with a maximum iteration count of 128 and a minimum iteration count of 1. If the iteration count is outside the range for the directive, results are unpredictable. The compiler will diagnose misuse at compile time (when able) or at run time if option `-h dir_check` is specified.

The syntax of these directives are as follows:

```
#pragma _CRI shortloop
#pragma _CRI shortloop128
```

The `shortloop` and `shortloop128` directives are exactly equivalent to `#pragma _CRI loop_info min_trips(1) max_trips(64)` and `#pragma _CRI loop_info min_trips(1) max_trips(128)`, respectively. The `loop_info` pragma is the preferred form.

The following examples illustrate the use of the `shortloop` and `shortloop128` directives:

```
#pragma _CRI shortloop
for (i = 0; i < n; i++) { /* 0 <= n <= 63 */
    a[i] = b[i] + c[i];
}

#pragma _CRI shortloop128
for (i = 0; i < n; i++) { /* 0 <= n <= 127 */
    a[i] = b[i] + c[i];
}
```

## 3.8 Scalar Directives

This section describes the scalar optimization directives, which control aspects of code generation, register storage, and other scalar operations.

### 3.8.1 `collapse` and `nocollapse` Directives

Scope:            Local

The loop collapse directives control collapse of the immediately following loop nest.

The formats of these directives are as follows:

```
#pragma _CRI collapse loop-number1,loop-number2[,loop-number3]...
#pragma _CRI nocollapse
```

When the `collapse` directive is applied to a loop nest, the loop numbers of the participating loops must be listed in order of increasing access stride. Loop numbers range from 1 to the nesting level of the most deeply nested loop. The directive enables the compiler to assume appropriate conformity between trip counts. The compiler diagnoses misuse at compile time (when able); or, if `-h dir_check` is specified, at run time.

The `nocollapse` directive disqualifies the immediately following loop from collapsing with any other loop. Collapse is almost always desirable, so use this directive sparingly.

Loop collapse is a special form of loop coalesce. Any perfect loop nest may be coalesced into a single loop, with explicit rediscovery of the intermediate values of original loop control variables. The rediscovery cost, which generally involves integer division, is quite high. Therefore, coalesce is rarely suitable for vectorization. It may be beneficial for multithreading.

By definition, loop collapse occurs when loop coalesce may be done without the rediscovery overhead. To meet this requirement, all memory accesses must have uniform stride.

### 3.8.2 concurrent Directive

Scope:           Local

The `concurrent` directive indicates that no data dependence exists between array references in different iterations of the loop that follows the directive. This can be useful for vectorization optimizations.

The format of the `concurrent` directive is as follows:

```
#pragma _CRI concurrent [safe_distance=n]
```

*n*                   An integer that represents the number of additional consecutive loop iterations that can be executed in parallel without danger of data conflict. *n* must be an integral constant > 0.

The `concurrent` directive is ignored if the `safe_distance` clause is used and vectorization is requested on the command line.

In the following example, the `concurrent` directive indicates that the relationship  $k > 3$  is true. The compiler will safely load all the array references `x[i-k]`, `x[i-k+1]`, `x[i-k+2]`, and `x[i-k+3]` during loop iteration *i*.

```
#pragma _CRI concurrent safe_distance=3

for (i = k + 1; i < n; i++) {
    x[i] = a[i] + x[i-k];
}
```

### 3.8.3 interchange and nointerchange Directives

Scope:            Local

The loop interchange control directives specify whether or not the order of the following two or more loops should be interchanged. These directives apply to the loops that they immediately precede.

The formats of these directives are as follows:

```
#pragma _CRI interchange(loop_number1, loop_number2[, loop_number3] ...)  
#pragma _CRI nointerchange
```

The first format specifies two or more loop numbers. Loop numbers range from 1 to nesting depth of the most deeply nested loop. They can be specified in any order, and the compiler reorders the loops. The loops must be perfectly nested. If the loops are not perfectly nested, you may receive unexpected results. The compiler reorders the loops such that the loop with *loop-number1* is outermost, then *loop-number2*, then *loop-number3*.

The second format inhibits loop interchange on the loop that immediately follows the directive.

In the following example, the `interchange` directive reorders the loops; the `k` loop becomes the outermost and the `i` loop the innermost:

```
#define N 100  
  
A[N][N][N];  
  
void  
f(int n)  
{  
    int i, j, k;  
  
    #pragma _CRI interchange( 2, 3, 1 )  
    for (i=0; i < n; i++) {  
        for (k=0; k < n; k++) {  
            for (j = 0; j < n; j++) {  
                A[k][j][i] = 1.0;  
            }  
        }  
    }  
}
```

### 3.8.4 noreduction Directive

**Note:** This directive is no longer recognized. Use the `#pragma _CRI novector` directive instead.

### 3.8.5 suppress Directive

The `suppress` directive suppresses optimization in two ways, determined by its use with either global or local scope.

The global scope `suppress` directive specifies that all associated local variables are to be written to memory before a call to the specified function. This ensures that the value of the variables will always be current.

The global `suppress` directive takes the following form:

```
#pragma _CRI suppress func...
```

The local scope `suppress` directive stores current values of the specified variables in memory. If the directive lists no variables, all variables are stored to memory.

This directive causes the values of these variables to be reloaded from memory at the first reference following the directive.

The local `suppress` directive has the following format:

```
#pragma _CRI suppress [var] ...
```

The net effect of the local `suppress` directive is similar to declaring the affected variables to be volatile except that the `volatile` qualifier affects the entire program, whereas the local `suppress` directive affects only the block of code in which it resides.

### 3.8.6 [no]unroll Directive

Scope:            Local

The `unroll` directive allows the user to control unrolling for individual loops or to specify no unrolling of a loop. Loop unrolling can improve program performance by revealing cross-iteration memory optimization opportunities such as read-after-write and read-after-read. The effects of loop unrolling also include:

- Improved loop scheduling by increasing basic block size
- Reduced loop overhead
- Improved chances for cache hits

The formats for these compiler directives are:

```
#pragma _CRI unroll n
#pragma _CRI nounroll
```

The `nounroll` directive disables loop unrolling for the next loop and does not accept the integer argument *n*. The `nounroll` directive is equivalent to the `unroll 0` and `unroll 1` directives.

The *n* argument applies only to the `unroll` directive and specifies no loop unrolling ( $n = 0$  or  $1$ ) or the total number of loop body copies to be generated ( $2 \leq n \leq 63$ ).

If you do not specify a value for  $n$ , the compiler will determine the number of copies to generate based on the number of statements in the loop nest.

**Note:** The compiler cannot always safely unroll non-innermost loops due to data dependencies. In these cases, the directive is ignored (see [Example 10](#)).

The `unroll` directive can be used only on loops with iteration counts that can be calculated before entering the loop. If `unroll` is specified on a loop that is not the innermost loop in a loop nest, the inner loops must be nested perfectly. That is, all loops in the nest can contain only one loop, and the innermost loop can contain work.

### Example 9. Unrolling outer loops

In the following example, assume that the outer loop of the following nest will be unrolled by 2:

```
#pragma _CRI unroll 2
for (i = 0; i < 10; i++) {
    for (j = 0; j < 100; j++) {
        a[i][j] = b[i][j] + 1;
    }
}
```

With outer loop unrolling, the compiler produces the following nest, in which the two bodies of the inner loop are adjacent:

```
for (i = 0; i < 10; i += 2) {
    for (j = 0; j < 100; j++) {
        a[i][j] = b[i][j] + 1;
    }
    for (j = 0; j < 100; j++) {
        a[i+1][j] = b[i+1][j] + 1;
    }
}
```

The compiler then *jams*, or *fuses*, the inner two loop bodies, producing the following nest:

```
for (i = 0; i < 10; i += 2) {
    for (j = 0; j < 100; j++) {
        a[i][j] = b[i][j] + 1;
        a[i+1][j] = b[i+1][j] + 1;
    }
}
```



**Example 10. Illegal unrolling of outer loops**

Outer loop unrolling is not always legal because the transformation can change the semantics of the original program. For example, unrolling the following loop nest on the outer loop would change the program semantics because of the dependency between `a[i][...]` and `a[i+1][...]`:

```
/* directive will cause incorrect code due to dependencies! */
#pragma _CRI unroll 2
for (i = 0; i < 10; i++) {
    for (j = 1; j < 100; j++) {
        a[i][j] = a[i+1][j-1] + 1;
    }
}
```

**3.8.7 [no]fusion Directive**

Scope:            Local

The `nofusion` directive instructs the compiler to not attempt loop fusion on the following loop even when the `-h fusion` option was specified on the compiler command line. The `fusion` directive instructs the compiler to attempt loop fusion on the following loop unless `-h nofusion` was specified on the compiler command line.

The formats for these compiler directives are:

```
#pragma _CRI fusion
#pragma _CRI nofusion
```

**3.9 Inlining Directives**

Inlining replaces calls to user-defined functions with the code that represents the function. This can improve performance by saving the expense of the function call overhead. It also increases the possibility of additional code optimization. Inlining may increase object code size.

Inlining is invoked in the following ways:

- Automatic inlining is enabled by the `-h ipan` option alone, as described in [Inlining Optimization Options on page 40](#).
- Explicit inlining is enabled by the `-h ipafrom=source [:source]` option alone as described in `-h ipafrom=source [source] ...` on [page 42](#).
- Combined inlining is enabled by using both the `-h ipan` and `-h ipafrom=source[:source]` options (see [Combined Inlining on page 43](#)).

Inlining directives can only appear in local scope; that is, inside a function definition. Inlining directives always take precedence over the command line settings.

The `-h report=i` option writes messages identifying where functions are inlined or briefly explains why functions are not inlined.

### 3.9.1 `clone_enable`, `clone_disable`, `clone_reset` Directives

The `clone_enable` and `clone_disable` directives control whether cloning is attempted over a range of code.

If `clone_enable` is in effect, cloning is attempted at call sites. If `clone_disable` is in effect, cloning is not attempted at call sites.

The `clone_reset` directive resets cloning to the state specified on the compiler command line.

These directives have the following formats:

```
#pragma _CRI clone_enable
#pragma _CRI clone_disable
#pragma _CRI clone_reset
```

One of these directives remains in effect until the opposite directive is encountered, until the end of the program unit, or until the `clone_reset` directive is encountered. These directives are ignored if the `-h ipa0` option is in effect.

### 3.9.2 `inline_enable`, `inline_disable`, and `inline_reset` Directives

The `inline_enable` pragma directs the compiler to attempt to inline functions at call sites. It has the following format:

```
#pragma _CRI inline_enable
```

The `inline_disable` directive tells the compiler to not inline functions at call sites. It has the following format:

```
#pragma _CRI inline_disable
```

The `inline_reset` directive returns the inlining state to the state specified on the command line (`-h ipan`). It has the following format:

```
#pragma _CRI inline_reset
```

The `inline_always` directive specifies functions that the compiler should always attempt to inline. If the directive is placed in the definition of the function, inlining is attempted at every call site to *name* in the entire input file being compiled. If the directive is placed in a function other than the definition, inlining is attempted at every call site to *name* within the specific function containing the directive.

The `inline_never` directive specifies functions that are never to be inlined. If the directive is placed in the definition of the function, inlining is never attempted at any call site to *name* in the entire input file being compiled. If the directive is placed in a function other than the definition, inlining is never attempted at any call site to *name* within the specific function containing the directive.

The `inline_always` and `inline_never` directives have the following formats:

```
#pragma _CRI inline_always name [,name ] ...
#pragma _CRI inline_never name [,name ] ...
```

The following example illustrates the use of these directives.

**Example 11. Using the `inline_enable`, `inline_disable`, and `inline_reset` directives**

To compile the file displayed in this example, enter the following commands:

```
% cc -hipa4 b.c
% cat b.c
void qux(int x)
{

void bar(void);
int a = 1;

    x = a+a+a+a+a+a+a+a+a+a+a;
    bar();
}

void foo(void)
{
int j = 1;

#pragma inline_enable          /* enable inlining at all call sites here forward */
    qux(j);
    qux(j);
#pragma inline_disable        /* disable inlining at all call sites here forward */
    qux(j);
#pragma inline_reset          /* reset control to the command line -hipa4 */
    qux(j);
}
```

**Example 12. Using `inline_reset`**

The following code fragment shows how the `#pragma _CRI inline_reset` directive would affect code compiled with the `-h ipa3` option:

```
{
void f1()
#pragma _CRI inline_disable /* No inlining will be done in f1;
    ...
}

void f2()
{
#pragma _CRI inline_disable /* turn off all inlining to the end of
the routine or another directive is encountered.
    ...

#pragma _CRI inline_reset /* The inlining state is -h ipa3
for the remainder of f2;
    ...
}
```

### 3.9.3 `inline_always` and `inline_never` Directives

The `inline_always` directive specifies functions that the compiler should always attempt to inline. If the directive is placed in the definition of the function, inlining is attempted at every call site to *name* in the entire input file being compiled. If the directive is placed in a function other than the definition, inlining is attempted at every call site to *name* within the specific function containing the directive.

The format of the `inline_always` directive is as follows:

```
#pragma _CRI inline_always name [,name ] ...
```

The `inline_never` directive specifies functions that are never to be inlined. If the directive is placed in the definition of the function, inlining is never attempted at any call site to *name* in the entire input file being compiled. If the directive is placed in a function other than the definition, inlining is never attempted at any call site to *name* within the specific function containing the directive.

The format of the `inline_never` directive is as follows:

```
#pragma _CRI inline_never name [,name ] ...
```

The *name* argument is the name of a function.

This chapter describes the Cray implementation of the OpenMP 3.0 standard (*OpenMP Application Program Interface Version 3.0 May 2008 Copyright © 1997-2008 OpenMP Architecture Review Board*).

## 4.1 Deferred OpenMP Features

The following OpenMP Fortran features are not yet supported by the Cray C and Cray C++ compilers:

- In C++, an object of a class with a nontrivial default constructor, a nontrivial copy constructor, a nontrivial destructor, or a nontrivial copy assignment operator can be used in a scoping clause. Currently, these objects can be used only in a shared scope.
- In C++, an object of a class with a nontrivial constructor, a nontrivial copy constructor, a nontrivial destructor, or a nontrivial copy assignment operator cannot be used in a scoping clause for any scope except shared. This limitation will be removed in a future release.
- In C++, random-access iterator loops marked for work sharing may not be work shared. You can help the compiler by using variables (such as `begin` and `end` in example 2 below) as snapshots that capture the initial values of the `begin` and `end` vectors (`vec.begin()` and `vec.end()`).

Iterator example 1 is not work shared:

```
void iterator_example1() {
#pragma omp parallel for default(none) shared(vec)
    for (it = vec.begin(); it < vec.end(); it++ ) {
        printf( "vector element is %i\n", *it );
    }
}
```

Iterator example 2 is work shared:

```
void iterator_example2() {
    begin = vec.begin();
    end = vec.end();
#pragma omp parallel for default(none) shared(vec,begin,end)
    for (it = begin; it < end; it+=1 ) {
        printf( "vector element is %i\n", *it );
    }
}
```

This limitation will be removed in a future release.

- Orphaned task constructs may have an implicit `taskwait` directive added to the end of the routine. This is not required by the specification but is currently required by the Cray implementation. This limits the amount of parallelism that may be seen. This limitation will be removed in a future release.
- Task switching is not implemented. The thread that starts executing a task will be the thread that finishes the thread. Task switching will be implemented in a future release.
- The `collapse` clause is accepted but is not implemented in the compiler. This limitation will be removed in a future release.

## 4.2 Cray Implementation Differences

The OpenMP C and C++ Application Program Interface specification defines areas of implementation that have vendor-specific behaviors. The following sections describe Cray-specific implementations.

### 4.2.1 Pragmas

#### 4.2.1.1 `atomic` Construct

The `atomic` directive is replaced with a `critical` section that encloses the statement.

#### 4.2.1.2 `for` Construct

For the `schedule(guided, chunk)` clause, the size of the initial chunk for the master thread and other team members is approximately equal to the trip count divided by the number of threads.

For the `schedule(runtime)` clause, the schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable. If this environment variable is not set, the schedule type and chunk size default to `static` and 0, respectively.

In the absence of the `schedule` clause, the default schedule is `static` and the default chunk size is approximately the number of iterations divided by the number of threads.

#### 4.2.1.3 `parallel` Construct

If a parallel region is encountered while dynamic adjustment of the number of threads is disabled, and the number of threads specified for the parallel region exceeds the number that the runtime system can supply, the program terminates.

The number of physical processors actually hosting the threads at any given time is fixed at program startup and is specified by the `aprun -d depth` option.

The `OMP_NESTED` environment variable and the `omp_set_nested()` call control nested parallelism. To enable nesting, set `OMP_NESTED` to `true` or use the `omp_set_nested()` call. Nesting is disabled by default.

#### 4.2.1.4 `private` Clause

If a variable is declared as `private`, the variable is referenced in the definition of a statement function, and the statement function is used within the lexical extent of the directive construct, then the statement function references the `private` version of the variable.

#### 4.2.1.5 `threadprivate` Construct

The `threadprivate` directive specifies that variables are replicated, with each thread having its own copy. If the dynamic threads mechanism is enabled, the definition and association status of a thread's copy of the variable is undefined, and the allocation status of an allocatable array is undefined.

### 4.2.2 OpenMP Library Routines

#### 4.2.2.1 `omp_get_max_active_levels()`

The Cray implementation of OpenMP supports the proposed OpenMP 3.0 `omp_get_max_active_levels()` routine to return the maximum number of nested parallel levels currently allowed.

#### 4.2.2.2 `omp_set_dynamic()`

The `omp_set_dynamic()` routine enables or disables dynamic adjustment of the number of threads available for the execution of subsequent parallel regions by setting the value of the *dyn-var* ICV. The default is `on`.

#### 4.2.2.3 `omp_set_schedule()`

The `omp_set_schedule()` routine affects the schedule that is applied when runtime is used as schedule kind, by setting the value of the *run-sched-var* ICV. The default is `on`.

#### 4.2.2.4 `omp_set_max_active_levels()`

The Cray implementation of OpenMP supports the proposed OpenMP 3.0 `omp_set_max_active_levels()` routine to limit the depth of nested parallelism. The number specified controls the maximum number of nested parallel levels with more than one thread. The default value is 1 (nesting disabled).

#### 4.2.2.5 `omp_set_nested()`

The `omp_set_nested()` routine enables or disables nested parallelism, by setting the *nest-var* ICV. The default is `false`.

#### 4.2.2.6 `omp_set_num_threads()`

If dynamic adjustment of the number of threads is disabled, the `number_of_threads_expr` argument sets the number of threads for all subsequent parallel regions until this procedure is called again with a different value.

### 4.2.3 OpenMP Environment Variables

#### 4.2.3.1 `OMP_DYNAMIC`

The default value is `true`.

#### 4.2.3.2 `OMP_MAX_ACTIVE_LEVELS`

The default value is 1.

#### 4.2.3.3 `OMP_NESTED`

The default value is `false`.

#### 4.2.3.4 `OMP_NUM_THREADS`

If this environment variable is not set and you do not use the `omp_set_num_threads()` routine to set the number of OpenMP threads, the default is 1 thread.

The maximum number of threads per compute node is 4 times the number of allocated processors. If the requested value of `OMP_NUM_THREADS` is more than the number of threads an implementation can support, the behavior of the program depends on the value of the `OMP_DYNAMIC` environment variable. If `OMP_DYNAMIC` is `false`, the program terminates. If `OMP_DYNAMIC` is `true`, it uses up to 4 times the number of allocated processors. For example, on a quad-core Cray XT system, this means the program can use up to 16 threads per compute node.

#### 4.2.3.5 `OMP_SCHEDULE`

The default values for this environment variable are `static` for *type* and 0 for *chunk*.

#### 4.2.3.6 `OMP_STACKSIZE`

The default value for this environment variable is 128 MB.



#### 4.2.3.7 OMP\_THREAD\_LIMIT

Sets the number of OpenMP threads to use for the entire OpenMP program by setting the *thread-limit-var* ICV. The Cray implementation defaults to 4 times the number of available processors.

#### 4.2.3.8 OMP\_WAIT\_POLICY

Provides a hint to an OpenMP implementation about the desired behavior of waiting threads by setting the *wait-policy-var* ICV. A compliant OpenMP implementation may or may not abide by the setting of the environment variable. The default value for this environment variable is `active`.

### 4.3 Compiler Options Affecting OpenMP

These Cray C and C++ Compiler options affect OpenMP applications:

- `-h [no]omp` ([-h \[no\]omp on page 58](#))
- `-h threadn` ([-h threadn on page 36](#))

### 4.4 OpenMP Program Execution

The `aprun -d` option can be used to define the number of processors available to the application. If neither the `OMP_NUM_THREADS` environment variable nor the `omp_set_num_threads()` call is used to set the number of OpenMP threads, the system defaults to 1 thread. The maximum number of threads per compute node is 4 times the number of allocated processors.

For further information, including example OpenMP programs, see the *Cray XT Programming Environment User's Guide*.



# Using Cray Unified Parallel C (UPC) [5]

---

Unified Parallel C (UPC) is a C language extension for parallel program development.

Cray supports the UPC Language Specification 1.2 and also supports some Cray specific functions as noted in the following sections. For additional information about a function, refer to the appropriate UPC man page. For a description of the `-h upc` command line option, see [Compiling and Executing UPC Code on page 100](#).

You should be familiar with UPC and understand the differences between the published UPC Introduction and Language Specification paper and the current UPC specification. If you are not familiar with UPC, refer to the UPC home page at <http://upc.gwu.edu/>. Under the Publications link, select the *Introduction to UPC and Language Specification* paper. This paper is slightly outdated but contains valuable information about understanding and using UPC. The UPC home page also contains, under the Documentation link, the *UPC Language Specification 1.2* paper.

UPC allows you to explicitly specify parallel programming through language syntax rather than library functions such as those used in MPI and SHMEM by allowing you to read and write memory of other processes with simple assignment statements. Program synchronization occurs only when explicitly programmed; there is no implied synchronization. These methods map very well onto Cray XT systems and enable users to achieve high performance.

**Note:** UPC is a dialect of the C language. It is not available in C++.

UPC allows you to maintain a view of your program as a collection of threads operating in a common global address space without burdening you with details of how parallelism is implemented on the machine (for example, as shared memory or as a collection of physically distributed memories).

UPC data objects are private to a single thread or shared among all threads of execution. Each thread has a unique memory space that holds its private data objects, and access to a globally-shared memory space that is distributed across the threads. Thus, every part of a shared data object has an affinity to a single thread.

Cray UPC is compatible with MPI and SHMEM.

**Note:** UPC 1.2 supports a parallel I/O model which provides control over file synchronization. However, if you continue to use the regular C I/O routines, you must supply the controls as needed to remove race conditions. File I/O under UPC is very similar to standard C because one thread opens a file and shares the file handle, and multiple threads may read or write to the same file.

**Note:** Some UPC constructs perform more efficiently than others. For more information about UPC functions, see the man pages (read `intro_upc(3)` first).

## 5.1 Cray Implementation Differences

There is a false sharing hazard when referencing shared `char` and `short` integers.

If two PEs store a `char` or `short` to the same 64-bit word in memory without synchronization, incorrect results can occur. It is possible for one PE's store to be lost. This is because these stores are implemented by reading the entire 64-bit word, inserting the `char` or `short` value and writing the entire word back to memory.

The following output is a result of two PEs writing two different characters into the same word in memory without synchronization:

	Register	Memory
Initial Value		0x0000
PE 0 Reads	0x0000	0x0000
PE 1 Reads	0x0000	0x0000
PE 0 Inserts 3	0x3000	0x0000
PE 1 Inserts 7	0x0700	0x0000
PE 0 Writes	0x3000	0x3000
PE 1 Writes	0x0700	0x0700

Notice that the value stored by PE 0 has been lost. The final value intended was 0x3700. This situation is referred to as false sharing. It is the result of supporting data types that are smaller than the smallest type that can be individually read or written by the hardware. UPC programmers must take care when storing to shared `char` and `short` data that this situation does not occur.

## 5.2 Compiling and Executing UPC Code

To compile UPC code, you must load the programming environment module and specify the `-h upc` option on the `cc` command line.

The `-X npes` option can optionally be used to define the number of threads to use and statically set the value of the `THREADS` constant. See [-X npes on page 59](#) for requirements regarding the use of the `-X npes` option.

Dynamic shared memory allocated through language features is not done through `shmallloc()`; it is done by the run time system. The run time system checks the `XT_SYMMETRIC_HEAP_SIZE` environment variable. If the variable is not set, the default (32 MB) is used. The default may change in a future release, so it is good practice to define `XT_SYMMETRIC_HEAP_SIZE` if your program uses a significant amount of dynamically allocated shared data.

If your program requires a total amount of shared memory (static plus dynamic) that is near or more than 2 GB per PE, then you may encounter a GASNet limit. The Cray XT UPC runtime uses GASNet, which has a default shared segment size of 2 GB. This default may be inappropriate on nodes with 4 GB per PE, for example. The workaround is to set `GASNET_MAX_SEGSIZE=4GB`.

### **Example 13. UPC and THREADS defined dynamically**

The following example enables UPC and allows the `THREADS` symbol to be defined dynamically for the `exampl` application:

```
% cc -h upc -o multupc exampl.c
```

### **Example 14. UPC and THREADS defined statically**

The following example enables UPC and statically defines the `THREADS` symbol as 15 for the `exampl` application:

```
% cc -h upc -x15 -o multupc exampl.c
```

The processing elements specified by *npes* are compute node cores.

After compiling the UPC code, you run the program using the `aprun` command.

If you use the `-X npes` compiler option, you must specify the same number of threads in the `aprun` command.



# Using Cray C++ Libraries [6]

---

The Cray C++ compiler supports the C++ 98 standard (ISO/IEC FDIS 14882) and continues to support existing Cray extensions. Most of the standard C++ features are supported, except for the few mentioned in [Unsupported Standard C++ Library Features](#).

For information about C++ language conformance and exceptions, see [Appendix B, Using Cray C and C++ Dialects on page 151](#).

## 6.1 Unsupported Standard C++ Library Features

The Cray C++ compiler supports the C++ standard except for wide characters and multiple locales as follows:

- String classes using basic string class templates with wide character types or that use the `wstring` standard template class
- I/O streams using wide character objects
- File-based streams using file streams with wide character types (`wfilebuf`, `wifstream`, `wofstream`, and `wfstream`)
- Multiple localization libraries; Cray C++ supports only one locale

**Note:** The C++ standard provides a standard naming convention for library routines. Therefore, classes or routines that use wide characters are named appropriately. For example, the `fscanf` and `sprintf` functions do not use wide characters, but the `fwscanf` and `swprintf` function do.





# Using Cray C++ Template Instantiation [7]

---

A *template* describes a class or function that is a model for a family of related classes or functions. The act of generating a class or function from a template is called *template instantiation*.

For example, a template can be created for a stack class, and then a stack of integers, a stack of floats, and a stack of some user-defined type can be used. In source code, these might be written as `Stack<int>`, `Stack<float>`, and `Stack<X>`. From a single source description of the template for a stack, the compiler can create instantiations of the template for each of the types required.

The instantiation of a class template is always done as soon as it is needed during a compilation. However, the instantiations of template functions, member functions of template classes, and static data members of template classes (template entities) are not necessarily done immediately for the following reasons:

- The preferred end result is one copy of each instantiated entity across all object files in a program. This applies to entities with external linkage.
- A specialization of a template entity is allowed. For example, a specific version of `Stack<int>`, or of just `Stack<int>::push` could be written to replace the template-generated version and to provide a more efficient representation for a particular data type.
- If a template function is not referenced, it should not be compiled because such functions could contain semantic errors that would prevent compilation. Therefore, a reference to a template class should not automatically instantiate all the member functions of that class.

The goal of an instantiation mode is to provide trouble-free instantiation. The programmer should be able to compile source files to object code, link them and run the resulting program, without questioning how the necessary instantiations are done.

In practice, this is difficult for a compiler to do, and different compilers use different instantiation schemes with different strengths and weaknesses.

The Cray C++ compiler requires a normal, top-level, explicitly compiled source file that contains the definition of both the template entity and of any types required for the particular instantiation. This requirement is met in one of the following ways:

- Each `.h` file that declares a template entity also contains either the definition of the entity or includes another file containing the definition.
- When the compiler identifies a template declaration in a `.h` file and discovers a need to instantiate that entity, implicit inclusion gives the compiler permission to search for an associated definition file having the same base name and a different suffix and implicitly include that file at the end of the compilation (see [Implicit Inclusion on page 112](#)).
- The programmer makes sure that the files that define template entities also have the definitions of all the available types and adds code or directives in those files to request instantiation of those entities.

The Cray C++ compiler provides two instantiation mechanisms—simple instantiation and prelinker instantiation. These mechanisms perform template instantiation and provide command line options and `#pragma` directives that give the programmer more explicit control over instantiation.

## 7.1 Simple Instantiation

The goal of the simple instantiation mode is to provide a method of instantiating templates without the need to create and manage intermediate (`*.ti` and `*.ii`) files.

The Cray C++ compilers accomplishes simple instantiation as follows:

1. When the source files of a program are compiled using the `-h simple_templates` option, each of the `*.o` files contains a copy of all of the template instantiations it uses.
2. When the object files are linked together, the resulting executable file contains multiple copies of the template function.

Unlike in prelinker instantiation, no `*.ti` or `*.ii` files are created. The programmer is not required to manage the naming and location of the intermediate files.

The simple template instantiation process creates slightly larger object files and a slightly larger executable file than is the case for prelinker instantiation.

For example, you have three C++ source files, `x.C`, `y.C`, and `z.C`. The source files reference a template `sortall` that sorts `int`, `float`, and `char` array elements:

```
template <class X> void sortall(X a[])
{
    ... code to sort int, float, char elements ...
}
```

Entering the command `CC -c -h simple_templates x.C y.C z.C` produces object files `x.o`, `y.o`, and `z.o`. Each `*.o` file has three copies of `sortall`, one for ints, one for floats, and one for chars.

Then, entering the command `CC x.o y.o z.o` links the files and any needed library routines, creating `a.out`.

Because the `-h simple_templates` option enables the `-h instantiate=used` option, all needed template entities are instantiated. The programmer can use the `#pragma do_not_instantiate` directive in programs compiled using the `-h simple_templates` option. For more information, see [Instantiation Directives on page 76](#).

## 7.2 Prelinker Instantiation

In prelinker mode, automatic instantiation is accomplished by the Cray C++ compiler as follows:

1. If the compiler is responsible for doing all instantiations automatically, it can only do so for the entire program. That is, the compiler cannot make decisions about instantiation of template entities until all source files of the complete program have been read.
2. The first time the source files of a program are compiled, no template entities are instantiated. However, the generated object files contain information about things that could have been instantiated in each compilation. For any source file that makes use of a template instantiation, an associated `.ti` file is created, if one does not already exist (for example, the compilation of `abc.C` results in the creation of `abc.ti`).

3. When the object files are linked together, a program called the *prelinker* is run. It examines the object files, looking for references and definitions of template entities and for any additional information about entities that could be instantiated.



**Caution:** The prelinker examines the object files in a library (.a) file but, because it does not modify them, is not able to assign template instantiations to them.

4. If the prelinker finds a reference to a template entity for which there is no definition in the set of object files, it looks for a file that indicates that it could instantiate that template entity. Upon discovery of such a file, it assigns the instantiation to that file. The set of instantiations assigned to a given file (for example, abc.C) is recorded in an associated file that has a .ii suffix (for example, abc.ii).
5. The prelinker then executes the compiler to again recompile each file for which the .ii was changed.
6. During compilation, the compiler obeys the instantiation requests contained in associated .ii file and produces a new object file that contains the requested template entities and the other things that were already in the object file.
7. The prelinker repeats steps 3 through 5 until there are no more instantiations to be adjusted.
8. The object files are linked together.

Once the program has been linked correctly, the .ii files contain a complete set of instantiation assignments. If source files are recompiled, the compiler consults the .ii files and does the indicated instantiations as it does the normal compilations. That means that, except in cases where the set of required instantiations changes, the prelink step from then on will find that all the necessary instantiations are present in the object files and no instantiation assignment adjustments need be done. This is true even if the entire program is recompiled. Because the .ii file contains information about how to recompile when instantiating, it is important that the .o and .ii files are not moved between the first compilation and linkage.

The prelinker cannot instantiate into and from library files (.a), so if a library is to be shared by many applications its templates should be expanded. You may find that creating a directory of objects with corresponding .ii files and the use of `-h prelink_copy_if_nonlocal` (see [-h prelink\\_copy\\_if\\_nonlocal \(CC\) on page 29](#)) will work as if you created a library (.a) that is shared.

The `-h prelink_local_copy` option indicates that only local files (for example, files in the current directory) are candidates for assignment of instantiations. This option is useful when you are sharing some common relocatables but do not want them updated. Another way to ensure that shared .o files are not updated is to use the `-h remove_instantiation_flags` option when compiling the shared .o files. This also makes smaller resulting shared .o files.

An easy way to create a library that instantiates all references of templates within the library is to create an empty main function and link it with the library, as shown in the following example. The prelinker will instantiate those template references that are within the library to one of the relocatables without generating duplicates. The empty `dummy_main.o` file is removed prior to creating the `.a` file.

```
% CC a.C b.C c.C dummy_main.C
% ar cr mylib.a a.o b.o c.o
```

Another alternative to creating a library that instantiates all references of templates is to use the `-h one_instantiation_per_object` option. This option directs the prelinker to instantiate each template referenced within a library in its own object file. The following example shows how to use the option:

```
% CC -h one_instantiation_per_object a.C b.C c.C dummy_main.C
% ar cr mylib.a a.o b.o c.o myInstantiationsDir/*.int.o
```

For more information about this alternative see [One Instantiation Per Object File on page 110](#) and [-h one\\_instantiation\\_per\\_object \(CC\) on page 27](#).

Prelinker instantiation can coexist with partial explicit control of instantiation by the programmer through the use of `#pragma` directives or the `-h instantiate=mode` option.

Prelinker instantiation mode can be disabled by issuing the `-h noautoinstantiate` command line option. If prelinker instantiation is disabled, the information about template entities that could be instantiated in a file is not included in the object file.

## 7.3 Instantiation Modes

Normally, during compilation of a source file, no template entities are instantiated (except those assigned to the file by prelinker instantiation). However, the overall instantiation mode can be changed by issuing the `-h instantiate=mode` command line option. The *mode* argument can be specified as follows:

<u>mode</u>	<u>Description</u>
<code>none</code>	Do not automatically create instantiations of any template entities. This is the most appropriate mode when prelinker instantiation is enabled. This is the default instantiation mode.
<code>used</code>	Instantiate those template entities that were used in the compilation. This includes all static data members that have template definitions.
<code>all</code>	Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members are instantiated, regardless of whether they were used. Nonmember template functions are instantiated even if the only reference was a declaration.
<code>local</code>	Similar to <code>used</code> mode, except that the functions are given internal linkage. This mode provides a simple mechanism for those who are not familiar with templates. The compiler instantiates the functions used in each compilation unit as local functions, and the program links and runs correctly (barring problems due to multiple copies of local static variables). This mode may generate multiple copies of the instantiated functions and is not suitable for production use. This mode cannot be used in conjunction with prelinker template instantiation. Prelinker instantiation is disabled by this mode.

In the case where the `CC` command is given a single source file to compile and link, all instantiations are done in the single source file and, by default, the `used` mode is used and prelinker instantiation is suppressed.

## 7.4 One Instantiation Per Object File

You can direct the prelinker to instantiate each template referenced in the source into its own object file. This method is preferred over other template instantiation object file generation options because:

- The user of a library pulls in only the instantiations that are needed.
- Multiple libraries with the same template can link. If each instantiation is not placed in its own object file, linking a library with another library that also contains the same instantiations will generate warnings on some platforms.

Use the `-h one_instantiation_per_object` option to generate one object file per instantiation. For more information about this option, see [-h one\\_instantiation\\_per\\_object \(CC\) on page 27](#).

## 7.5 Instantiation `#pragma` Directives

Instantiation `#pragma` directives can be used in source code to control the instantiation of specific template entities or sets of template entities. There are three instantiation `#pragma` directives:

- The `#pragma _CRI instantiate` directive causes a specified entity to be instantiated.
- The `#pragma _CRI do_not_instantiate` directive suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition is supplied.
- The `#pragma _CRI can_instantiate` directive indicates that a specified entity can be instantiated in the current compilation, but need not be. It is used in conjunction with prelinker instantiation to indicate potential sites for instantiation if the template entity is deemed to be required by the compiler.

The argument to the `#pragma _CRI instantiate` directive can be any of the following:

- A template class name. For example: `A<int>`
- A template class declaration. For example: `class A<int>`
- A member function name. For example: `A<int>::f`
- A static data member name. For example: `A<int>::i`
- A static data declaration. For example: `int A<int>::i`
- A member function declaration. For example: `void A<int>::f(int, char)`
- A template function declaration. For example: `char* f(int, float)`

A `#pragma` directive in which the argument is a template class name (for example, `A<int>` or `class A<int>`) is equivalent to repeating the directive for each member function and static data member declared in the class. When instantiating an entire class, a given member function or static data member may be excluded using the `#pragma _CRI do_not_instantiate` directive. For example:

```
#pragma _CRI instantiate A<int>
#pragma _CRI do_not_instantiate A<int>::f
```

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by use of the `#pragma _CRI instantiate` directive and no template definition is available or a specific definition is provided, an error is issued.

The following example illustrates the use of the `#pragma _CRI instantiate` directive:

```
template <class T> void f1(T); // No body provided
template <class T> void g1(T); // No body provided
void f1(int) {} // Specific definition
void main()
{
    int    i;
    double d;
    f1(i);
    f1(d);
    g1(i);
    g1(d);
}
#pragma _CRI instantiate void f1(int) // error-specific definition
#pragma _CRI instantiate void g1(int) // error-no body provided
```

In the preceding example, `f1(double)` and `g1(double)` are not instantiated because no bodies are supplied, but no errors will be produced during the compilation. If no bodies are supplied at link time, a linker error is issued.

A member function name (such as `A<int>::f`) can be used as a `#pragma` directive argument only if it refers to a single, user-defined member function (that is, not an overloaded function). Compiler-generated functions are not considered, so a name can refer to a user-defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration, as in the following example:

```
#pragma _CRI instantiate char* A<int>::f(int, char*)
```

The argument to an instantiation directive cannot be a compiler-generated function, an inline function, or a pure virtual function.

## 7.6 Implicit Inclusion

The implicit inclusion feature implies that if the compiler needs a definition to instantiate a template entity declared in a `.h` file, it can implicitly include the corresponding `.C` file to get the source code for the definition. For example, if a template entity `ABC::f` is declared in file `xyz.h`, and an instantiation of `ABC::f` is required in a compilation, but no definition of `ABC::f` appears in the source code processed by the compilation, the compiler will search for the `xyz.C` file and, if it exists, process it as if it were included at the end of the main source file.



To find the template definition file for a given template entity, the Cray C++ compiler must know the full path name to the file in which the template was declared and whether the file was included using the system include syntax (such as `#include <file.h>`). This information is not available for preprocessed source code containing `#line` directives. Consequently, the Cray C++ compiler does not attempt implicit inclusion for source code that contains `#line` directives.

The definition-file suffixes that are tried by default are `.c`, `.C`, `.cxx`, `.CXX`, and `.cc`.

Implicit inclusion works well with prelinker instantiation; however, they are independent. They can be enabled or disabled independently, and implicit inclusion is still useful without prelinker instantiation.



# Using Cray C Extensions [8]

---

The Cray C compiler supports the following Cray extensions to the C standard:

- Complex data extensions ([Complex Data Extensions on page 115](#))
- `fortran` keyword ([fortran Keyword on page 116](#))
- Hexadecimal floating-point constants ([Hexadecimal Floating-point Constants on page 116](#))

A program that uses one or more extensions does not strictly conform to the standard. These extensions are not available in strict conformance mode.

## 8.1 Complex Data Extensions

Cray C extends the complex data facilities defined by standard C with these extensions:

- Imaginary constants
- Incrementing or decrementing `_Complex` data

The Cray C compiler supports the Cray imaginary constant extension and is defined in the `<complex.h>` header file. This imaginary constant has the following form:

$Ri$

$R$  is either a floating constant or an integer constant; no space or other character can appear between  $R$  and  $i$ . If you are compiling in strict conformance mode (`-h conform`), the Cray imaginary constants are not available.

The following example illustrates imaginary constants:

```
#include <complex.h>
double complex z1 = 1.2 + 3.4i;
double complex z2 = 5i;
```

The other extension to the complex data facility allows the prefix- and postfix-increment and decrement operators to be applied to the `_Complex` data type. The operations affect only the real portion of a complex number.

## 8.2 `fortran` Keyword

In extended mode, the identifier `fortran` is treated as a keyword. It specifies a storage class that can be used to declare a Fortran-coded external function. The use of the `fortran` keyword when declaring a function causes the compiler to verify that the arguments used in each call to the function are pass by addresses; any arguments that are not addresses are converted to addresses.

As in any function declaration, an optional *type-specifier* declares the type returned, if any. Type `int` is the default; type `void` can be used if no value is returned (by a Fortran subroutine). The `fortran` storage class causes conversion of lowercase function names to uppercase, and, if the function name ends with an underscore character, the trailing underscore character is stripped from the function name. (Stripping the trailing underscore character is in keeping with UNIX practice.)

Functions specified with a `fortran` storage class must not be declared elsewhere in the file with a `static` storage class.

**Note:** The `fortran` keyword is not allowed in Cray C++.

An example using the `fortran` keyword is shown in [Cray C and Fortran Example on page 135](#).

## 8.3 Hexadecimal Floating-point Constants

The Cray C compiler supports the standard hexadecimal floating constant notations and the Cray hexadecimal floating constant notation. The standard hexadecimal floating constants are portable and have sizes that are dependent upon the hardware. The remainder of this section discusses the Cray hexadecimal floating constant.

The Cray hexadecimal floating constant feature is not portable, because identical hexadecimal floating constants can have different meanings on different systems. It can be used whenever traditional floating-point constants are allowed.

The hexadecimal constant has the usual syntax: `0x` (or `0X`) followed by hexadecimal characters. The optional floating suffix has the same form as for normal floating constants: `f` or `F` (for float), `l` or `L` (for long), optionally followed by an `i` (imaginary).

The constant must represent the same number of bits as its type, which is determined by the suffix (or the default of double). The constant's bit length is four times the number of hexadecimal digits, including leading zeros.

The following example illustrates hexadecimal constant representation:

```
0x7f7fffff.f
```

32-bit float

```
0x0123456789012345.
```

64-bit double

The value of a hexadecimal floating constant is interpreted as a value in the specified floating type. This uses an unsigned integral type of the same size as the floating type, regardless of whether an object can be explicitly declared with such a type. No conversion or range checking is performed. The resulting floating value is defined in the same way as the result of accessing a member of floating type in a union after a value has been stored in a different member of integral type.

The following example illustrates hexadecimal floating-point constant representation that use Cray floating-point format:

```
int main(void)
{
    float f1, f2;
    double g1, g2;

    f1 = 0x3ec00000.f;
    f2 = 0x3fc00000.f;
    g1 = 0x40fa400100000000.;
    g2 = 0x40fa400200000000.;

    printf("f1 = %8.8g\n", f1);
    printf("f2 = %8.8g\n", f2);
    printf("g1 = %16.16g\n", g1);
    printf("g2 = %16.16g\n", g2);
    return 1;
}
```

This is the output for the previous example:

```
f1 =    0.375
f2 =     1.5
g1 =  107520.0625
g2 =  107520.125
```



# Using Predefined Macros [9]

---

The macros listed in this chapter are the Cray-specific predefined macros. To see the entire list of predefined macros, add `-Wp,-list_final_macros -E` to your `cc` command line. For example, if you have the file `c.c`, specify:

```
% cc -Wp,-list_final_macros -E c.c > out
```

Predefined macros can be divided into the following categories:

- Macros required by the C and C++ standards ([Macros Required by the C and C++ Standards on page 120](#))
- Macros based on the host machine ([Macros Based on the Host Machine on page 120](#))
- Macros based on the target machine ([Macros Based on the Target Machine on page 121](#))
- Macros based on the compiler ([Macros Based on the Compiler on page 121](#))
- UPC macros ([UPC Predefined Macros on page 122](#))

Predefined macros provide information about the compilation environment. In this chapter, only those macros that begin with the underscore (`_`) character are defined when running in strict-conformance mode.

**Note:** Any of the predefined macros except those required by the standard (see [Macros Required by the C and C++ Standards on page 120](#)) can be undefined by using the `-U` command line option; they can also be redefined by using the `-D` command line option.

A large set of macros is also defined in the standard header files.

## 9.1 Macros Required by the C and C++ Standards

The following macros are required by the C and C++ standards:

Macro	Description
<code>__TIME__</code>	Time of translation of the source file.
<code>__DATE__</code>	Date of translation of the source file.
<code>__LINE__</code>	Line number of the current line in your source file.
<code>__FILE__</code>	Name of the source file being compiled.
<code>__STDC__</code>	Defined as the decimal constant 1 if compilation is in strict conformance mode; defined as the decimal constant 2 if the compilation is in extended mode. This macro is defined for Cray C and C++ compilations.
<code>__cplusplus</code>	Defined as 1 when the compiling Cray C++ code and undefined when compiling Cray C code. The <code>__cplusplus</code> macro is required by the ISO C++ standard, but not the ISO C standard.

## 9.2 Macros Based on the Host Machine

The following macros provide information about the environment running on the host machine:

Macro	Description
<code>__linux</code>	Defined as 1.
<code>__linux__</code>	Defined as 1.
<code>linux</code>	Defined as 1.
<code>__gnu_linux__</code>	Defined as 1.



## 9.3 Macros Based on the Target Machine

The following macros provide information about the characteristics of the target machine:

Macro	Description
<code>_ADDR64</code>	Defined as 1 if the targeted CPU has 64-bit address registers; if the targeted CPU does not have 64-bit address registers, the macro is not defined.
<code>__LITTLE_ENDIAN__</code>	Defined as 1. Cray XT systems use little endian byte ordering.
<code>_LITTLE_ENDIAN</code>	Defined as 1. Cray XT systems use little endian byte ordering.
<code>_MAXVL_8</code>	Defined as 16, the number of 8-bit elements that fit in an XMM register ("vector length").
<code>_MAXVL_16</code>	Defined as 8.
<code>_MAXVL_32</code>	Defined as 4.
<code>_MAXVL_64</code>	Defined as 2.
<code>_MAXVL_128</code>	Defined as 0.

## 9.4 Macros Based on the Compiler

The following macros provide information about compiler features:

Macro	Description
<code>_RELEASE</code>	Defined as the major release level of the compiler.
<code>_RELEASE_MINOR</code>	Defined as the minor release level of the compiler.
<code>_RELEASE_STRING</code>	Defined as a string that describes the version of the compiler.
<code>_CRAYC</code>	Defined as 1 to identify the Cray C and C++ compilers.

## 9.5 UPC Predefined Macros

The following macros provide information about UPC functions:

Macro	Description
<code>__UPC__</code>	The integer constant 1, indicating a conforming implementation.
<code>__UPC_DYNAMIC_THREADS__</code>	The integer constant 1 in the dynamic THREADS translation environment.
<code>__UPC_STATIC_THREADS__</code>	The integer constant 1 in the static THREADS translation environment.

# Running C and C++ Applications [10]

---

To run applications, log in to a Cray XT login node and set up your user environment. See the *Cray XT Programming Environment User's Guide* for details on setting up your environment. In your working directory, load the appropriate modules, compile your programs, and launch them using the `aprun` command.

**Note:** You need to be in a Lustre-mounted directory, such as `/lus/nid00007/user1/myxtapps`, before using the `aprun` command.

To use the Cray C compiler, load the `PrgEnv-cray` module. Use the `module list` command to get a list of currently loaded modules. If another Programming Environment module is loaded, use the `module swap` command. For example, if `PrgEnv-pgi` is loaded, use this command:

```
% module swap PrgEnv-pgi PrgEnv-cray
```

Then use the `cc -V` command to verify that the Cray C compiler is available:

```
% cc -V
/opt/cray/xt-asyncpe/2.5/bin/cc: INFO: native target is being used
Cray C++ : Version 7.1.0.129 Thu May 21, 2009 14:37:25
```

Compile and run your application.

```
% cc -o simple simple.c
% aprun -n 4 ./simple | sort
Application 1024906 resources: utime 0, stime 0
hello from pe 0 of 4
hello from pe 1 of 4
hello from pe 2 of 4
hello from pe 3 of 4
```

If you specified the `-X` option on the `cc` command line, then the `aprun -n` option must specify the same number of processing elements (npes).

For additional information, see the *Cray XT Programming Environment User's Guide*.



# Debugging Cray C and C++ Code [11]

---

The TotalView symbolic debugger is available to help you debug C and C++ codes (see *TotalView Users Guide*). In addition, the Cray C and C++ compilers provide the following features to help you in debugging codes:

- The `-G` and `-g` compiler options provide symbol information about your source code for use by the TotalView debugger. For more information about these compiler options, see [-G level and -g on page 47](#).
- The `-h [no]bounds` option and the `#pragma _CRI [no]bounds` directive let you check pointer and array references. The `-h [no]bounds` option is described in [-h \[no\]bounds \(cc\) on page 48](#). The `#pragma _CRI [no]bounds` directive is described in [\[no\]bounds Directive on page 68](#).
- The `#pragma _CRI message` directive lets you add warning messages to sections of code where you suspect problems. The `#pragma _CRI message` directive is described in [message Directive on page 70](#).
- The `#pragma _CRI [no]opt` directive lets you selectively isolate portions of your code to optimize, or to toggle optimization on and off in selected portions of your code. The `#pragma _CRI [no]opt` directive is described in [\[no\]opt Directive on page 72](#).

## 11.1 TotalView Debugger

Some of the functions available in the TotalView debugger allow you to perform the following actions:

- Set and clear breakpoints, which can be conditional, at both the source code level and the assembly code level
- Examine core files
- Step through a program, including across function calls
- Reattach to the executable file after editing and recompiling
- Edit values of variables and memory locations
- Evaluate code fragments

## 11.2 Compiler Debugging Options

Compiler options control the trade-offs between ease of debugging and compiler optimizations. The compiler produces internal debugger information (DWARF) at all times. The DWARF data provides function and line information to debuggers for tracebacks and breakpoints, as well as type and location information about data variables.

These options are specified as follows:

- `-Gf`

With no DWARF, the executable is optimized and as small as possible, but cannot be easily debugged. Only assembly instructions will be visible and only global symbols will be available.

- `-Gp`

With partial DWARF and at least some optimization, tracebacks and limited breakpoints are available in the debugger. The source code will be visible and many more symbols will be available. The executable will be somewhat slower and larger in exchange for increased debugger functionality.

- `-g` or `-Gn`

With full DWARF and no optimizations, full debugging will be available, but at the cost of a slower and larger executable.

**Note:** The `-g/-G` options may be specified on a per file basis so that only part of an application incurs the overhead of improved debugging.

However, consider the following cases in which optimization is affected by the `-Gp` and `-Gf` debugging options:

- Vectorization can be inhibited if a label exists within the vectorizable loop.
- Vectorization can be inhibited if the loop contains a nested block and the `-Gp` option is specified.
- When the `-Gp` option is specified, setting a breakpoint at the first statement in a vectorized loop allows you to stop and display at each vector iteration. However, setting a breakpoint at the first statement in an unrolled loop may not allow you to stop at each vector iteration.

# Using Interlanguage Communication [12]

---

In some situations, it is necessary or advantageous to make calls to assembly or Fortran functions from C or C++ programs. This chapter describes how to make such calls. It also discusses calls to C and C++ functions from Fortran and assembly language. For additional information about interlanguage communication, see *Interlanguage Programming Conventions*. The calling sequence is described in detail on the `callseq(3)` man page.

The C and C++ compilers provide a mechanism for declaring external functions that are written in other languages. This allows you to write portions of an application in C, C++, Fortran, or assembly language. This can be useful in cases where the other languages provide performance advantages or utilities that are not available in C or C++.

This chapter describes how to call assembly language and Fortran programs from a C or C++ program. It also discusses the issues related to calling C or C++ programs from other languages.

## 12.1 Calls between C and C++ Functions

The following requirements must be considered when making calls between functions written in C and C++:

- In Cray C++, the `extern "C"` linkage is required when declaring an external function that is written in Cray C or when declaring a Cray C++ function that is to be called from Cray C. Normally the compiler will mangle function names to encode information about the function's prototype in the external name. This prevents direct access to these function names from a C function. The `extern "C"` keyword will prevent the compiler from performing name mangling.
- The program must be linked using the `CC` command.
- The program's main routine must be C or C++ code compiled with the `CC` command.

Objects can be shared between C and C++. There are some Cray C++ objects that are not accessible to Cray C functions (such as classes). The following object types can be shared directly:

- Integral and floating types.
- Structures and unions that are declared identically in C and C++. In order for structures and unions to be shared, they must be declared with identical members in the identical order.
- Arrays and pointers to the above types.

In the following example, a Cray C function (`C_add_func`) is called by the Cray C++ main program:

```
#include <iostream.h>

extern "C" int C_add_func(int, int);
int global_int = 123;

main()
{
    int res, i;

    cout << "Start C++ main" << endl;

    /* Call C function to add two integers and return result. */

    cout << "Call C C_add_func" << endl;
    res = C_add_func(10, 20);
    cout << "Result of C_add_func = " << res << endl;
    cout << "End C++ main" << endl;
}
```

The Cray C function (`C_add_func`) is as follows:

```
#include <stdio.h>

extern int global_int;

int C_add_func(int p1, int p2)
{
    printf("\tStart C function C_add_func.\n");
    printf("\t\t p1      = %d\n", p1);
    printf("\t\t p2      = %d\n", p2);
    printf("\t\t global_int = %d\n", global_int);
    return p1 + p2;
}
```



The output from the execution of the calling sequence illustrated in the preceding example is as follows:

```
Start C++ main
Call C C_add_func
    Start C function C_add_func.
        p1      = 10
        p2      = 20
        global_int = 123
Result of C_add_func = 30
End C++ main
```

## 12.2 Calling Fortran Functions and Subroutines from C or C++

This subsection describes the following aspects of calling Fortran from C or C++. Topics include requirements and guidelines, argument passing, array storage, logical and character data, accessing named common, and accessing blank common.

### 12.2.1 Requirements

Keep the following points in mind when calling Fortran functions from C or C++:

- Fortran uses the call-by-address convention. C and C++ use the call-by-value convention, which means that only pointers should be passed to Fortran subprograms. For more information, see [Argument Passing on page 130](#).
- Fortran arrays are in column-major order. C and C++ arrays are in row-major order. This indicates which dimension is indicated by the first value in an array element subscript. For more information, see [Array Storage on page 130](#).
- Single-dimension arrays of signed 32-bit integers and single-dimension arrays of 32-bit floating-point numbers are the only aggregates that can be passed as parameters without changing the arrays.
- Fortran character pointers and character pointers from Cray C and C++ are incompatible. For more information, see [Logical and Character Data on page 131](#).
- Fortran logical values and the Boolean values from C and C++ are not fully compatible. For more information, see [Logical and Character Data on page 131](#).
- External C and C++ variables are stored in common blocks of the same name, making them readily accessible from Fortran programs if the C or C++ variable is in uppercase.
- When declaring Fortran functions or objects in C or C++, the name must be specified in all uppercase letters, digits, or underscore characters and consist of 31 or fewer characters.
- In Cray C, Fortran functions can be declared using the `fortran` keyword (see [fortran Keyword on page 116](#)). The `fortran` keyword is not available in

Cray C++. Instead, Fortran functions must be declared by specifying `extern "C"`.

## 12.2.2 Argument Passing

Because Fortran subroutines expect arguments to be passed by pointers rather than by value, C and C++ functions called from Fortran subroutines must pass pointers rather than values.

All argument passing in Cray C is strictly by value. To prepare for a function call between two Cray C functions, a copy is made of each actual argument. A function can change the values of its formal parameters, but these changes cannot affect the values of the actual arguments. It is possible, however, to pass a pointer. (All array arguments are passed by this method.) This capability is analogous to the Fortran method of passing arguments.

In addition to passing by value, Cray C++ also provides passing by reference.

## 12.2.3 Array Storage

C and C++ arrays are stored in memory in row-major order. Fortran arrays are stored in memory in column-major order. For example, the C or C++ array declaration `int A[3][2]` is stored in memory as:

A[0][0]	A[0][1]
A[1][0]	A[1][1]
A[2][0]	A[2][1]

The previously defined array is viewed linearly in memory as:

A[0][0] A[0][1] A[1][0] A[1][1] A[2][0] A[2][1]

The Fortran array declaration `INTEGER A(3,2)` is stored in memory as:

A(1,1)	A(2,1)	A(3,1)
A(1,2)	A(2,2)	A(3,2)

The previously defined array is viewed linearly in memory as:

A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2)

When an array is shared between Cray C, C++, and Fortran, its dimensions are declared and referenced in C and C++ in the opposite order in which they are declared and referenced in Fortran. Arrays are zero-based in C and C++ and are one-based in Fortran, so in C and C++ you should subtract 1 from the array subscripts that you would normally use in Fortran.

For example, using the Fortran declaration of array A in the preceding example, the equivalent declaration in C or C++ is:

```
int a[2][3];
```

The following list shows how to access elements of the array from Fortran and from C or C++:

<u>Fortran</u>	<u>C or C++</u>
A(1,1)	A[0][0]
A(2,1)	A[0][1]
A(3,1)	A[0][2]
A(1,2)	A[1][0]
A(2,2)	A[1][1]
A(3,2)	A[1][2]

## 12.2.4 Logical and Character Data

Logical and character data need special treatment for calls between C or C++ and Fortran. Fortran has a character descriptor that is incompatible with a character pointer in C and C++. The techniques used to represent logical (Boolean) values also differ between Cray C, C++, and Fortran.

Mechanisms you can use to convert one type to the other are provided by the `fortran.h` header file and conversion macros shown in the following list:

<u>Macro</u>	<u>Description</u>
<code>_btol</code>	Conversion utility that converts a 0 to a Fortran logical <code>.FALSE.</code> and a nonzero value to a Fortran logical <code>.TRUE.</code>
<code>_ltob</code>	Conversion utility that converts a Fortran logical <code>.FALSE.</code> to a 0 and a Fortran logical <code>.TRUE.</code> to a 1.

## 12.2.5 Accessing Named Common from C and C++

The following example demonstrates how external C and C++ variables are accessible in Fortran named common blocks. It shows a C or C++ function calling a Fortran subprogram, the associated Fortran subprogram, and the associated input and output.

In this example, the C or C++ structure `_st` is accessed in the Fortran subprogram as common block `ST`. The Fortran common block `ST` will be converted to lower case with a trailing underscore added.

The name of the structure and the converted Fortran common block name must match. The C and C++ structure member names and the Fortran common block member names do not have to match, as is shown in this example.

The following Cray C main program calls the Fortran subprogram `FCTN`:

```
#include <stdio.h>
struct
{
    int i;
    double a[10];
    long double d;
} _st;

main()
{
    int i;

    /* initialize struct _st */
    _st.i = 12345;

    for (i = 0; i < 10; i++)
        _st.a[i] = i;

    _st.d = 1234567890.1234567890L;

    /* print out the members of struct _st */
    printf("In C: _st.i = %d, _st.d = %20.10Lf\n", _st.i, _st.d);
    printf("In C: _st.a = ");
    for (i = 0; i < 10; i++)
        printf("%4.1f", _st.a[i]);
    printf("\n\n");

    /* call the fortran function */
    FCTN();
}
```

The following example is the Fortran subprogram FCTN called by the previous Cray C main program:

```
C ***** Fortran subprogram (f.f): *****

      SUBROUTINE FCTN

      COMMON /ST/STI, STA(10), STD
      INTEGER STI
      REAL STA
      DOUBLE PRECISION STD

      INTEGER I

      WRITE(6,100) STI, STD
100  FORMAT ('IN FORTRAN: STI = ', I5, ', STD = ', D25.20)
      WRITE(6,200) (STA(I), I = 1,10)
200  FORMAT ('IN FORTRAN: STA =', 10F4.1)
      END
```

The previous Cray C and Fortran examples are executed by the following commands, and they produce the output shown:

```
% cc -c c.c
% ftn -c f.f
% ftn c.o f.o
% ./a.out
ST.i = 12345, ST.d = 1234567890.1234567890
In C: ST.a = 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

IN FORTRAN: STI = 12345, STD = .12345678901234567889D+10
IN FORTRAN: STA = 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
```

## 12.2.6 Accessing Blank Common from C or C++

Fortran includes the concept of a common block. A *common block* is an area of memory that can be referenced by any program unit in a program. A *named common block* has a name specified in names of variables or arrays stored in the block. A *blank common block*, sometimes referred to as blank common, is declared in the same way, but without a name.

There is no way to access blank common from C or C++ similar to accessing a named common block. However, you can write a simple Fortran function to return the address of the first word in blank common to the C or C++ program and then use that as a pointer value to access blank common.

The following example shows how Fortran blank common can be accessed using C or C++ source code:

```
#include <stdio.h>

struct st
{
    float a;
    float b[10];
```

```
    } *ST;

#ifdef __cplusplus
    extern "C" struct st *MYCOMMON(void);
    extern "C" void FCTN(void);
#else
    fortran struct st *MYCOMMON(void);
    fortran void FCTN(void);
#endif

main()
{
    int i;

    ST = MYCOMMON();
    ST->a = 1.0;
    for (i = 0; i < 10; i++)
        ST->b[i] = i+2;
    printf("\n In C and C++\n");
    printf("      a = %5.1f\n", ST->a);
    printf("      b = ");
    for (i = 0; i < 10; i++)
        printf("%5.1f ", ST->b[i]);
    printf("\n\n");

    FCTN();
}
```

This Fortran source code accesses blank common and is accessed from the C or C++ source code in the preceding example:

```
SUBROUTINE FCTN
COMMON // STA,STB(10)
PRINT *, "IN FORTRAN"
PRINT *, "      STA = ",STA
PRINT *, "      STB = ",STB
STOP
END

FUNCTION MYCOMMON()
COMMON // A
MYCOMMON = LOC(A)
RETURN
END
```

This is the output of the previous C or C++ source code:

```
a =  1.0
b =  2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0  10.0  11.0
```

This is the output of the previous Fortran source code:

```
STA = 1.
STB = 2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,  10.,  11.
```

## 12.2.7 Cray C and Fortran Example

Here is an example of a Cray C function that calls a Fortran subprogram. The Fortran subprogram example follows the Cray C function example, and the input and output from this sequence follows the Fortran subprogram example.

**Note:** This example assumes that the Cray Fortran function is compiled with the `-s default32` option enabled. The examples will not work if the `-s default64` option is enabled.

```
/*                      C program (main.c):                      */

#include <stdio.h>
#include <string.h>
#include <fortran.h>

/* Declare prototype of the Fortran function. Note the last */
/* argument passes the length of the first argument. */
fortran double FTNFCTN (char *, int *, int);

double FLOAT1 = 1.6;
double FLOAT2; /* Initialized in FTNFCTN */

main()
{
    int clogical, ftnlogical, cstringlen;
    double rtnval;
    char *cstring = "C Character String";

    /* Convert clogical to its Fortran equivalent */
    clogical = 1;
    ftnlogical = _btol(clogical);

    /* Print values of variables before call to Fortran function */
    printf(" In main: FLOAT1 = %g; FLOAT2 = %g\n",
           FLOAT1, FLOAT2);
    printf(" Calling FTNFCTN with arguments:\n");
    printf(" string = \"%s\"; logical = %d\n\n", cstring, clogical);
    cstringlen = strlen(cstring);
    rtnval = FTNFCTN(cstring, &ftnlogical, cstringlen);

    /* Convert ftnlogical to its C equivalent */
    clogical = _ltob(&ftnlogical);

    /* Print values of variables after call to Fortran function */
    printf(" Back in main: FTNFCTN returned %g\n", rtnval);
    printf(" and changed the two arguments:\n");
    printf(" string = \"%.*s\"; logical = %d\n",
           cstringlen, cstring, clogical);
}
```

```
C          Fortran subprogram (ftnfctn.f):

FUNCTION FTNFCTN(STR, LOG)

REAL FTNFCTN
CHARACTER*(*) STR
LOGICAL LOG

COMMON /FLOAT1/FLOAT1
COMMON /FLOAT2/FLOAT2
REAL FLOAT1, FLOAT2
DATA FLOAT2/2.4/          ! FLOAT1 INITIALIZED IN MAIN

C      PRINT CURRENT STATE OF VARIABLES
PRINT*, '      IN FTNFCTN: FLOAT1 = ', FLOAT1,
1      ';FLOAT2 = ', FLOAT2
PRINT*, '      ARGUMENTS:   STR = "', STR, '"'; LOG = ', LOG

C      CHANGE THE VALUES FOR STR(ING) AND LOG(ICAL)
STR = 'New Fortran String'
LOG = .FALSE.

FTNFCTN = 123.4
PRINT*, '      RETURNING FROM FTNFCTN WITH ', FTNFCTN
PRINT*
RETURN
END
```

The previous Cray C function and Fortran subprogram are executed by the following commands and produce the following output:

```
% cc -c main.c
% ftn -c ftnfctn.f
% ftn main.o ftnfctn.o
% ./a.out
In main: FLOAT1 = 1.6;  FLOAT2 = 2.4
Calling FTNFCTN with arguments:
string = "C Character String"; logical = 1

IN FTNFCTN: FLOAT1 = 1.6; FLOAT2 = 2.4
ARGUMENTS:   STR = "C Character String"; LOG = T
RETURNING FROM FTNFCTN WITH 123.4
Back in main: FTNFCTN returned 123.4
and changed the two arguments:
string = "New Fortran String"; logical = 0
```



## 12.2.8 Calling a Fortran Program from Cray C++

The following example illustrates how a Fortran program can be called from a Cray C++ program:

```
#include <iostream.h>
extern "C" int FORTRAN_ADD_INTS(int *arg1, int &arg2);

main()
{
    int num1, num2, res;
    cout << "Start C++ main" << endl << endl;

    //Call FORTRAN function to add two integers and return result.
    //Note that the second argument is a reference parameter so
    //it is not necessary to take the address of the
    //variable num2.

    num1 = 10;
    num2 = 20;
    cout << "Before Call to FORTRAN_ADD_INTS" << endl;
    res = FORTRAN_ADD_INTS(&num1, num2);
    cout << "Result of FORTRAN Add = " << res << endl << endl;
    cout << "End C++ main" << endl;
}
```

The Fortran program that is called from the Cray C++ main function in the preceding example is as follows:

```
INTEGER FUNCTION FORTRAN_ADD_INTS(Arg1, Arg2)
INTEGER Arg1, Arg2

PRINT *, " FORTRAN_ADD_INTS, Arg1,Arg2 = ", Arg1, Arg2
FORTRAN_ADD_INTS = Arg1 + Arg2
END
```

The output from the execution of the preceding example is as follows:

```
Start C++ main

Before Call to FORTRAN_ADD_INTS
  FORTRAN_ADD_INTS, Arg1,Arg2 =  10,  20
Result of FORTRAN Add = 30

End C++ main
```

## 12.3 Calling a C or C++ Function from Fortran

A C or C++ function can be called from a Fortran program. One of two methods can be used to call C functions from Fortran: the C interoperability feature provided by the Fortran 2000 facility or the method documented in this section. C interoperability provides a standard portable interoperability mechanism for Fortran and C programs. For more information about C interoperability, see the *Cray Fortran Reference Manual*. If you are using the method documented in this section to call C functions from Fortran, keep in mind the information in [Calling Fortran Functions and Subroutines from C or C++ on page 129](#).

When calling a Cray C++ function from a Fortran program, observe the following rules:

- The Cray C++ function must be declared with `extern "C"` linkage.
- The program must be linked with the `CC` command.
- The program's main routine must be C or C++ code compiled with the `CC` command.

The example that follows illustrates a Fortran program, `main.f`, that calls a Cray C function, `ctctn.c`. The Cray C function being called, the commands required, and the associated input and output are also included.

**Note:** This example assumes that the Cray Fortran program is compiled with the `-s default32` option enabled. The examples will not work if the `-s default64` option is enabled.

### Example 15. Calling a C function from Fortran

Fortran program `main.f` source code:

```
C  Fortran program (main.f):

      PROGRAM MAIN

      REAL CFCTN
      COMMON /FLOAT1/FLOAT1
      COMMON /FLOAT2/FLOAT2
      REAL FLOAT1, FLOAT2
      DATA FLOAT1/1.6/ ! FLOAT2 INITIALIZED IN cfctn.c
      LOGICAL LOG
      CHARACTER*24 STR
      REAL RTNVAL

C  INITIALIZE VARIABLES STR(ING) AND LOG(ICAL)
      STR = 'Fortran Character String'
      LOG = .TRUE.

C  PRINT VALUES OF VARIABLES BEFORE CALL TO C FUNCTION
      PRINT*, 'In main.f: FLOAT1 = ', FLOAT1,
1         ' ; FLOAT2 = ', FLOAT2
      PRINT*, 'Calling cfctn.c with these arguments: '
      PRINT*, 'LOG = ', LOG
```

```

        PRINT*, 'STR = ', STR

        RTNVAL = CFCTN(STR, LOG)

C  PRINT VALUES OF VARIABLES AFTER CALL TO C FUNCTION
        PRINT*, 'Back in main.f:: cfctn.c returned ', RTNVAL
        PRINT*, 'and changed the two arguments to: '
        PRINT*, 'LOG = ', LOG
        PRINT*, 'STR = ', STR

        END PROGRAM

```

Compile main.f, creating main.o:

```
% ftn -c main.f
```

C function cfctn.c source code:

```

/*  C function (cfctn.c)  */
#include <fortran.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

float FLOAT1; /* Initialized in MAIN */
float FLOAT2 = 2.4;

/* The slen argument passes the length of string in str */
float cfctn_(char * str, int *log, int slen)
{
    int clog;
    float rtnval;
    char *cstring;

    /* Convert log passed from Fortran MAIN */
    /* into its C equivalent */
    cstring = malloc(slen+1);
    strncpy(cstring, str, slen);
    cstring[slen] = '\0';
    clog = _ltob(log);

    /* Print the current state of the variables */
    printf(" In CFCTN: FLOAT1 = %.1f; FLOAT2 = %.1f\n",
           FLOAT1, FLOAT2);
    printf(" Arguments: str = '%s'; log = %d\n",
           cstring, clog);

    /* Change the values for str and log */
    strncpy(str, "C Character String ", 24);
    *log = 0;

    rtnval = 123.4;
    printf(" Returning from CFCTN with %.1f\n\n", rtnval);
    return(rtnval);
}

```

Compile cfctn.c, creating cfctn.o:

```
% cc -c cfctn.c
```

Link main.o and cfctn.o, creating executable interlang1:

```
% ftn -o interlang1 main.o cfctn.o
```

Run program interlang1:

```
% ./interlang1
```

Program output:

```
In main.f: FLOAT1 = 1.60000002 ; FLOAT2 = 2.4000001
Calling cfctn.c with these arguments:
LOG = T
STR = Fortran Character String
In CFCTN: FLOAT1 = 1.6; FLOAT2 = 2.4
Arguments: str = 'Fortran Character String'; log = 1
Returning from CFCTN with 123.4
```

```
Back in main.f:: cfctn.c returned 123.400002
and changed the two arguments to:
LOG = F
STR = C Character String
```

# Implementation-defined Behavior [13]

---

This chapter describes compiler behavior that is defined by the implementation according to the C and/or C++ standards. The standards require that the behavior of each particular implementation be documented.

The C and C++ standards define implementation-defined behavior as behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation. The behavior of the Cray C and C++ compilers for these cases is summarized in this chapter.

## 13.1 Messages

All diagnostic messages issued by the compilers are reported through the Cray Linux Environment (CLE) message system. For information about messages issued by the compilers and for information about the Cray Linux Environment (CLE) message system, see [Appendix C, Using the Compiler Message System on page 165](#).

## 13.2 Environment

When `argc` and `argv` are used as parameters to the `main` function, the array members `argv[0]` through `argv[argc-1]` contain pointers to strings that are set by the command shell. The shell sets these arguments to the list of words on the command line used to invoke the compiler (the argument list). For further information about how the words in the argument list are formed, refer to the documentation on the shell in which you are running. For information about Cray Linux Environment (CLE) shells, see the `sh(1)` or `csh(1)` man page.

A third parameter, `char **envp`, provides access to environment variables. The value of the parameter is a pointer to the first element of an array of null-terminated strings that matches the output of the `env(1)` command. The array of pointers is terminated by a null pointer.

The compiler does not distinguish between interactive devices and other, noninteractive devices. The library, however, may determine that `stdin`, `stdout`, and `stderr` (`cin`, `cout`, and `cerr` in Cray C++) refer to interactive devices and buffer them accordingly.

## 13.2.1 Identifiers

The *identifier* (as defined by the standards) is merely a sequence of letters and digits. Specific uses of identifiers are called *names*.

The Cray C compiler treats the first 255 characters of a name as significant, regardless of whether it is an internal or external name. The case of names, including external names, is significant. In Cray C++, all characters of a name are significant.

## 13.2.2 Types

[Table 12](#) summarizes Cray C and C++ types and the characteristics of each type. *Representation* is the number of bits used to represent an object of that type. *Memory* is the number of storage bits that an object of that type occupies.

In the Cray C and C++ compilers, *size*, in the context of the `sizeof` operator, refers to the size allocated to store the operand in memory; it does not refer to representation, as specified in [Table 12](#). Thus, the `sizeof` operator will return a size that is equal to the value in the *Memory* column of [Table 12](#) divided by 8 (the number of bits in a byte).

**Table 12. Data Type Mapping**

Type	Representation Size and Memory Storage Size (bits)
bool (C++)	8
_Bool (C)	8
char	8
wchar_t	32
short	16
int	32
long	64
long long	64
float	32
double	64
long double	64
float complex	64 (each part is 32 bits)
double complex	128 (each part is 64 bits)
long double complex	128 (each part is 64 bits)
Pointers	64

**Note:** Vectorization of 8- and 16-bit data types is deferred.

### 13.2.3 Characters

The full 8-bit ASCII code set can be used in source files. Characters not in the character set defined in the standard are permitted only within character constants, string literals, and comments. The `-h [no]calchars` option allows the use of the `$` sign in identifier names. For more information about the `-h [no]calchars` option, see [-h \[no\]calchars on page 32](#).

A character consists of 8 bits. Up to 8 characters can be packed into a 64-bit word. A plain `char` type (that is, one that is declared without a `signed` or `unsigned` keyword) is treated as a signed type.

Character constants and string literals can contain any characters defined in the 8-bit ASCII code set. The characters are represented in their full 8-bit form. A character constant can contain up to 8 characters. The integer value of a character constant is the value of the characters packed into a word from left to right, with the result right-justified, as shown in the following table:

**Table 13. Packed Characters**

Character constant	Integer value
'a'	0x61
'ab'	0x6162

In a character constant or string literal, if an escape sequence is not recognized, the `\` character that initiates the escape sequence is ignored, as shown in the following table:

**Table 14. Unrecognizable Escape Sequences**

Character constant	Integer value	Explanation
'\a'	0x7	Recognized as the ASCII BEL character
'\8'	0x38	Not recognized; ASCII value for 8
'\[ '	0x5b	Not recognized; ASCII value for [
'\c'	0x63	Not recognized; ASCII value for c

## 13.2.4 Wide Characters

Wide characters are treated as signed 64-bit integer types. Wide character constants cannot contain more than one multibyte character. Multibyte characters in wide character constants and wide string literals are converted to wide characters in the compiler by calling the `mbtowc()` function. The current locale in effect at the time of compilation determines the method by which `mbtowc()` converts multibyte characters to wide characters, and the shift states required for the encoding of multibyte characters in the source code. If a wide character, as converted from a multibyte character or as specified by an escape sequence, cannot be represented in the extended execution character set, it is truncated.

## 13.2.5 Integers

All integral values are represented in a two's complement format. For representation and memory storage requirements for integral types, see [Table 12](#).

When an integer is converted to a shorter signed integer, and the value cannot be represented, the result is the truncated representation treated as a signed quantity. When an unsigned integer is converted to a signed integer of equal length, and the value cannot be represented, the result is the original representation treated as a signed quantity.

The bitwise operators (unary operator `~` and binary operators `<<`, `>>`, `&`, `^`, and `|`) operate on signed integers in the same manner in which they operate on unsigned integers. The result of `e1 >> e2`, where `e1` is a negative-valued signed integral value, is that `e1` is right-shifted `e2` bit positions; vacated bits are filled with 1s. This behavior can be modified by using the `-h nosignedshifts` option (see [-h \[no\]signedshifts on page 32](#)). Bits higher than the sixth bit are not ignored.

The result of the `/` operator is the largest integer less than or equal to the algebraic quotient when either operand is negative and the result is a nonnegative value. If the result is a negative value, it is the smallest integer greater than or equal to the algebraic quotient. The `/` operator behaves the same way in C and C++ as in Fortran.

The sign of the result of the percent (`%`) operator is the sign of the first operand.

Integer overflow is ignored. Because some integer arithmetic uses the floating-point instructions, floating-point overflow can occur during integer operations. Division by 0 and all floating-point exceptions, if not detected as an error by the compiler, can cause a run time abort.

## 13.2.6 Arrays and Pointers

An unsigned `long` value can hold the maximum size of an array. The type `size_t` is defined to be a typedef name for unsigned `long` in the headers: `malloc.h`, `stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, and `time.h`. If more than one of these headers is included, only the first defines `size_t`.



A type `long` can hold the difference between two pointers to elements of the same array. The type `ptrdiff_t` is defined to be a `typedef` name for `long` in the header `stddef.h`.

If a pointer type's value is cast to a signed or unsigned `long int`, and then cast back to the original type's value, the two pointer values will compare equal.

Pointers on Cray Linux Environment (CLE) systems are byte pointers. Byte pointers use the same internal representation as integers; a byte pointer counts the numbers of bytes from the first address.

A pointer can be explicitly converted to any integral type large enough to hold it. The result will have the same bit pattern as the original pointer. Similarly, any value of integral type can be explicitly converted to a pointer. The resulting pointer will have the same bit pattern as the original integral type.

### 13.2.7 Registers

Use of the register storage class in the declaration of an object has no effect on whether the object is placed in a register. The compiler performs register assignment aggressively; that is, it automatically attempts to place as many variables as possible into registers.

### 13.2.8 Classes, Structures, Unions, Enumerations, and Bit Fields

Accessing a member of a union by using a member of a different type results in an attempt to interpret, without conversion, the representation of the value of the member as the representation of a value in the different type.

Members of a class or structure are packed into words from left to right. Padding is appended to a member to correctly align the following member, if necessary. Member alignment is based on the size of the member:

- For a member bit field of any size, alignment is any bit position that allows the member to fit entirely within a 64-bit word.
- For a member with a size less than 64 bits, alignment is the same as the size. For example, a `char` has a size and alignment of 8 bits; a `float` has a size and alignment of 32 bits.
- For a member with a size equal to or greater than 64 bits, alignment is 64 bits.
- For a member with array type, alignment is equal to the alignment of the element type.

A plain `int` type bit field is treated as a `signed int` bit field.

The values of an enumeration type are represented in the type `signed int` in C; they are a separate type in C++.

### 13.2.9 Qualifiers

When an object that has `volatile`-qualified type is accessed, it is simply a reference to the value of the object. If the value is not used, the reference need not result in a load of the value from memory.

### 13.2.10 Declarators

A maximum of 12 pointer, array, and/or function declarators are allowed to modify an arithmetic, structure, or union type.

### 13.2.11 Statements

The compiler has no fixed limit on the maximum number of case values allowed in a `switch` statement.

The Cray C++ compiler parses `asm` statements for correct syntax, but otherwise ignores them.

### 13.2.12 Exceptions

In Cray C++, when an exception is thrown, the memory for the temporary copy of the exception being thrown is allocated on the stack and a pointer to the allocated space is returned.

### 13.2.13 System Function Calls

For a description of the form of the unsuccessful termination status that is returned from a call to `exit(3)`, see the `exit(3)` man page.

## 13.3 Preprocessing

The value of a single-character constant in a constant expression that controls conditional inclusion matches the value of the same character in the execution character set. No such character constant has a negative value. For each, 'a' has the same value in the two contexts:

```
#if 'a' == 97
if ('a' == 97)
```

The `-I` option and the method for locating included source files is described in [-I \*incldir\* on page 54](#).

The source file character sequence in a `#include` directive must be a valid UNICOS/mp or Cray Linux Environment (CLE) file name or path name. A `#include` directive may specify a file name by means of a macro, provided the macro expands into a source file character sequence delimited by double quotes or `<` and `>` delimiters, as follows:

```
#define myheader "./myheader.h"
#include myheader
```

```
#define STDIO <stdio.h>
#include STDIO
```

The macros `__DATE__` and `__TIME__` contain the date and time of the beginning of translation. For more information, refer to the description of the predefined macros in [Chapter 9, Using Predefined Macros on page 119](#).

The `#pragma` directives are described in [Chapter 3, Using #pragma Directives on page 65](#).



# Using Libraries and the Loader [A]

---

This appendix describes the libraries that are available with the Cray C and C++ compilers and the loader.

## A.1 Cray C and C++ Libraries

Libraries that support Cray C and C++ are automatically available when you use the `CC` or `cc` command to compile your programs. These commands automatically issue the appropriate directives to load the program with the appropriate functions. If your program strictly conforms to the C or C++ standards, you do not need to know library names and locations. If your program requires other libraries or if you want direct control over the loading process, more knowledge of the loader and libraries is necessary.

The Standard Template Library (STL) is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template. Be sure you have a complete understanding of templates and how they work before using them.

## A.2 Loader

When you use the `cc` or `CC` command to invoke the compiler, and the program compiles without errors, the loader is called. Specifying the `-c` option on the command line produces relocatable object files (`*.o`) without calling the loader. These relocatable object files can then be used as input to the loader command by specifying the file names on the appropriate loader command line.

For example, the following command line compiles a file called `target.c` and produces the relocatable object file called `target.o` in your current working directory:

```
cc -c target.c
```

You can then use file `target.o` as input to the loader or save the file to use with other relocatable object files to compile and create a linked executable file (`a.out` by default).

Because of the special code needed to handle templates, constructors, destructors, and other C++ language features, object files generated by using the `CC` command should be linked using the `CC` command.

# Using Cray C and C++ Dialects [B]

---

This appendix details the features of the C and C++ languages that are accepted by the Cray C and C++ compilers, including certain language dialects and anachronisms. Users should be aware of these details, especially users who are porting codes from other environments.

## B.1 C++ Language Conformance

The Cray C++ compiler accepts the C++ language as defined by the *ISO/IEC 14882:1998* standard, with the exceptions listed in [Unsupported C++ Language Features on page 151](#).

The Cray C++ compiler also has a `cfront` compatibility mode, which duplicates a number of features and bugs of `cfront`. Complete compatibility is not guaranteed or intended. The mode allows programmers who have used `cfront` features to continue to compile their existing code (see [General Directives on page 67](#)). Command line options are also available to enable and disable anachronisms (see [C++ Anachronisms Accepted on page 155](#)) and strict standard-conformance checking (see [Extensions Accepted in Normal C++ Mode on page 156](#), and [Extensions Accepted in C or C++ Mode on page 157](#)). The command line options are described in [Chapter 2, Invoking the C and C++ Compilers on page 19](#).

### B.1.1 Unsupported C++ Language Features

The `export` keyword for templates is not supported. It is defined in the *ISO/IEC 14882:1998* standard, but is not in traditional C++.

### B.1.2 Supported C++ Language Features

The following features, which are in the *ISO/IEC 14882:1998* standard but not in traditional C++<sup>1</sup>, are supported:

- The dependent statement of an `if`, `while`, `do-while`, or `for` is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed.

---

<sup>1</sup> As defined in *The Annotated C++ Reference Manual (ARM)*, by Ellis and Stroustrup, Addison Wesley, 1990.

- The expression tested in an `if`, `while`, `do-while`, or `for`, as the first operand of a `?` operator, or as an operand of the `&&`, `||`, or `!` operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.
- Qualified names are allowed in elaborated type specifiers.
- A global-scope qualifier is allowed in member references of the form `x.::A::B` and `p->::A::B`.
- The precedence of the third operand of the `?` operator is changed.
- If control reaches the end of the `main()` routine, and the `main()` routine has an integral return type, it is treated as if a `return 0;` statement was executed.
- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.
- A functional-notation cast of the form `A()` can be used even if `A` is a class without a (nontrivial) constructor. The temporary that is created gets the same default initialization to zero as a static object of the class type.
- A cast can be used to select one out of a set of overloaded functions when taking the address of a function.
- Template friend declarations and definitions are permitted in class definitions and class template definitions.
- Type template parameters are permitted to have default arguments.
- Function templates may have nontype template parameters.
- A reference to `const volatile` cannot be bound to an rvalue.
- Qualification conversions such as conversion from `T**` to `T const * const` are allowed.
- Digraphs are recognized.
- Operator keywords (for example, `and` or `bitand`) are recognized.
- Static data member declarations can be used to declare member constants.
- `wchar_t` is recognized as a keyword and a distinct type.
- `bool` is recognized.
- RTTI (run time type identification), including `dynamic_cast` and the `typeid` operator, is implemented.
- Declarations in tested conditions (within `if`, `switch`, `for`, and `while` statements) are supported.
- Array `new` and `delete` are implemented.



- New-style casts (`static_cast`, `reinterpret_cast`, and `const_cast`) are implemented.
- Definition of a nested class outside its enclosing class is allowed.
- `mutable` is accepted on nonstatic data member declarations.
- Namespaces are implemented, including using declarations and directives. Access declarations are broadened to match the corresponding using declarations.
- Explicit instantiation of templates is implemented.
- The `typename` keyword is recognized.
- `explicit` is accepted to declare nonconverting constructors.
- The scope of a variable declared in the `for-init-statement` of a `for` loop is the scope of the loop (not the surrounding scope).
- Member templates are implemented.
- The new specialization syntax (using `template <>`) is implemented.
- Cv qualifiers are retained on `rvalues` (in particular, on function return values).
- The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between process overlay directives (PODs) and non-PODs with trivial constructors.
- The linkage specification is treated as part of the function type (affecting function overloading and implicit conversions).
- A `typedef` name can be used in an explicit destructor call.
- Placement delete is supported.
- An array allocated via a placement `new` can be deallocated via `delete`.
- `enum` types are considered to be nonintegral types.
- Partial specification of class templates is implemented.
- Partial ordering of function templates is implemented.
- Function declarations that match a function template are regarded as independent functions, not as “guiding declarations” that are instances of the template.
- It is possible to overload operators using functions that take `enum` types and no class types.
- Explicit specification of function template arguments is supported.
- Unnamed template parameters are supported.
- The new lookup rules for member references of the form `x.A::B` and `p->A::B` are supported.

- The notation `:: template` (and `->template`, etc.) is supported.
- In a reference of the form `f() ->g()`, with `g` a static member function, `f()` is evaluated. Likewise for a similar reference to a static data member. The ARM specifies that the left operand is not evaluated in such cases.
- `enum` types can contain values larger than can be contained in an `int`.
- Default arguments of function templates and member functions of class templates are instantiated only when the default argument is used in a call.
- String literals and wide string literals have `const` type.
- Class name injection is implemented.
- Argument-dependent (Koenig) lookup of function names is implemented.
- Class and function names declared only in unqualified friend declarations are not visible except for functions found by argument-dependent lookup.
- A `void` expression can be specified on a return statement in a `void` function.
- `reinterpret_cast` allows casting a pointer to a member of one class to a pointer to a member of another class even when the classes are unrelated.
- Two-phase name binding in templates as described in the *Working Paper* is implemented.
- Putting a `try/catch` around the initializers and body of a constructor is implemented.
- Template `template` parameters are implemented.
- Universal character set escapes (e.g., `\uabcd`) are implemented.
- `extern inline` functions are supported.
- Covariant return types on overriding virtual functions are supported.

## B.2 C++ Anachronisms Accepted

C++ anachronisms are enabled by using the `-h anachronisms` command line option (see [-h \[no\]anachronisms \(CC\) on page 23](#)). When anachronisms are enabled, the following anachronisms are accepted:

- `overload` is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized by using the default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.
- The number of elements in an array can be specified in an array delete operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name can be omitted in a base class initializer if there is only one immediate base class.
- Assignment to the `this` pointer in constructors and destructors is allowed. This is only allowed if anachronisms are enabled and the `assignment to this` configuration parameter is enabled.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as a non-nested class name if no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a non-`const` type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a non-`const` class type may be initialized from an `rvalue` of the class type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and can participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when checking for compatibility, therefore, the following statements declare the overloading of two functions named `f`:

```
int f(int);

int f(x) char x; { return x; }
```

**Note:** In C, this code is legal, but has a different meaning. A tentative declaration of `f` is followed by its definition.

## B.3 Extensions Accepted in Normal C++ Mode

The following C++ extensions are accepted (except when strict standard conformance mode is enabled, in which case a warning or caution message may be issued):

- A friend declaration for a class can omit the `class` keyword, as shown in the following example:

```
class B;
class A {
    friend B;    // Should be "friend class B"
};
```

- Constants of scalar type can be defined within classes, as shown in the following example:

```
class A {
    const int size=10;
    int a[size];
};
```

- In the declaration of a class member, a qualified name can be used, as shown in the following example:

```
struct A {
    int A::f();    // Should be int f();
}
```

- An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a “default” assignment operator; that is, such a declaration blocks the implicit generation of a copy assignment operator. This is `cfront` behavior that is known to be relied upon in at least one widely used library. Here is an example:

```
struct A { };
struct B : public A {
    B& operator=(A&);
};
```

By default, as well as in `cfront` compatibility mode, there will be no implicit declaration of `B::operator=(const B&)`, whereas in strict-ANSI mode, `B::operator=(A&)` is not a copy assignment operator and `B::operator=(const B&)` is implicitly declared.

- Implicit type conversion between a pointer to an extern "C" function and a pointer to an extern "C++" function is permitted. The following is an example:

```
extern "C" void f(); // f's type has extern "C" linkage
void (*pf)()        // pf points to an extern "C++" function
    = &f;           // error unless implicit conversion allowed
```

- The `?` operator, for which the second and third operands are string literals or wide string literals, can be implicitly converted to one of the following:

```
char *
wchar_t *
```

In C++ string literals are `const`. There is a deprecated implicit conversion that allows conversion of a string literal to `char *`, dropping the `const`. That conversion, however, applies only to simple string literals. Allowing it for the result of a `?` operation is an extension:

```
char *p = x ? "abc" : "def";
```

## B.4 Extensions Accepted in C or C++ Mode

The following extensions are accepted in C or C++ mode except when strict standard conformance modes is enabled, in which case a warning or caution message may be issued.

- The special `lint` comments `/*ARGSUSED*/`, `/*VARARGS*/` (with or without a count of nonvarying arguments), and `/*NOTREACHED*/` are recognized.
- A translation unit (input file) can contain no declarations.
- Comment text can appear at the ends of preprocessing directives.
- Bit fields can have base types that are `enum` or integral types in addition to `int` and `unsigned int`. This corresponds to A.6.5.8 in the ANSI Common Extensions appendix.
- `enum` tags can be incomplete as long as the tag name is defined and resolved by specifying the brace-enclosed list later.
- An extra comma is allowed at the end of an `enum` list.
- The final semicolon preceding the closing of a `struct` or `union` type specifier can be omitted.
- A label definition can be immediately followed by a right brace `( }`). (Normally, a statement must follow a label definition.)
- An empty declaration (a semicolon preceded by nothing) is allowed.
- An initializer expression that is a single value and is used to initialize an entire static array, `struct`, or `union` does not need to be enclosed in braces. ANSI C requires braces.
- In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it.
- The address of a variable with register storage class may be taken.

- In an integral constant expression, an integer constant can be cast to a pointer type and then back to an integral type.
- In duplicate size and sign specifiers (for example, `short short` or `unsigned unsigned`) the redundancy is ignored.
- Benign redeclarations of `typedef` names are allowed. That is, a `typedef` name can be redeclared in the same scope with the same type.
- Dollar sign (\$) characters can be accepted in identifiers by using the `-h calchars` command line option. This is not allowed by default.
- Numbers are scanned according to the syntax for numbers rather than the pp-number syntax. Thus, `0x123e+1` is scanned as three tokens instead of one token that is not valid. If the `-h conform` option is specified, the pp-number syntax is used.
- Assignment and pointer differences are allowed between pointers to types that are interchangeable but not identical, for example, `unsigned char *` and `char *`. This includes pointers to integral types of the same size (for example, `int *` and `long *`). Assignment of a string constant to a pointer to any kind of character is allowed without a warning.
- Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `const int **`). Comparisons and pointer difference of such pairs of pointer types are also allowed.
- In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ANSI C, these are allowed by some operators, and not by others (generally, where it does not make sense).
- Pointers to different function types may be assigned or compared for equality (`==`) or inequality (`!=`) without an explicit type cast. This extension is not allowed in C++ mode.
- A pointer to `void` can be implicitly converted to or from a pointer to a function type.
- External entities declared in other scopes are visible:

```
void f1(void) { extern void f(); }
void f2() { f(); /* Using out of scope declaration */ }
```
- In C mode, end-of-line comments (`//`) are supported.
- A non-lvalue array expression is converted to a pointer to the first element of the array when it is subscripted or similarly used.

- The `fortran` keyword. For more information, see [fortran Keyword on page 116](#).
- Cray hexadecimal floating point constants. For more information, see [Hexadecimal Floating-point Constants on page 116](#).

## B.5 C++ Extensions Accepted in `cfront` Compatibility Mode

The `cfront` compatibility mode is enabled by the `-h cfront` command-line option. The following extensions are accepted in `cfront` compatibility mode:

- Type qualifiers on the `this` parameter are dropped in contexts such as in the following example:

```
struct A {
    void f() const;
};
void (A::*fp)() = &A::f;
```

This is a safe operation. A pointer to a `const` function can be put into a pointer to non-`const`, because a call using the pointer is permitted to modify the object and the function pointed to will not modify the object. The opposite assignment would not be safe.

- Conversion operators that specify a conversion to `void` are allowed.
- A nonstandard `friend` declaration can introduce a new type. A `friend` declaration that omits the elaborated type specifier is allowed in default mode, however, in `cfront` mode the declaration can also introduce a new type name. An example follows:

```
struct A {
    friend B;
};
```

- The third operator of the `?` operator is a conditional expression instead of an assignment expression.
- A reference to a pointer type may be initialized from a pointer value without use of a temporary even when the reference pointer type has additional type qualifiers above those present in the pointer value. For example:

```
int *p;
const int *&r = p;    // No temporary used
```

- A reference can be initialized to `NULL`.
- Because `cfront` does not check the accessibility of types, access errors for types are issued as warnings instead of errors.

- When matching arguments of an overloaded function, a `const` variable with a value of 0 is not considered to be a null pointer constant. In general, in overload resolution, a null pointer constant must be entered as "0" to be considered a null pointer constant (e.g., '0' is not considered a null pointer constant).
- An alternate form of declaring pointer-to-member-function variables is supported, as shown in the following example:

```
struct A {  
    void f(int);  
    static void sf(int);  
    typedef void A::T3(int); // nonstd typedef decl  
    typedef void T2(int);    // std typedef  
};  
typedef void A::T(int);      // nonstd typedef decl  
T* pmf = &A::f;              // nonstd ptr-to-member decl  
A::T2* pf = A::sf;           // std ptr to static mem decl  
A::T3* pmf2 = &A::f;         // nonstd ptr-to-member decl
```

In this example, `T` is construed to name a function type for a nonstatic member function of class `A` that takes an `int` argument and returns `void`; the use of such types is restricted to nonstandard pointer-to-member declarations. The declarations of `T` and `pmf` in combination are equivalent to the following single standard pointer-to-member declaration:

```
void (A::* pmf)(int) = &A::f;
```

A nonstandard pointer-to-member declaration that appears outside of a class declaration, such as the declaration of `T`, is normally not valid and would cause an error to be issued. However, for declarations that appear within a class declaration, such as `A::T3`, this feature changes the meaning of a valid declaration. `cfront` version 2.1 accepts declarations, such as `T`, even when `A` is an incomplete type; so this case is also accepted.

- Protected member access checking is not done when the address of a protected member is taken. For example:

```
class B { protected: int i; };  
class D : public B { void mf();  
  
void D::mf() {  
    int B::* pm1 = &B::i; // error, OK in cfront mode  
    int D::* pm2 = &D::i; // OK  
}
```

**Note:** Protected member access checking for other operations (such as everything except taking a pointer-to-member address) is done normally.



- The destructor of a derived class can implicitly call the private destructor of a base class. In default mode, this is an error but in `cfront` mode it is reduced to a warning. For example:

```
class A {
    ~A();
};
class B : public A {
    ~B();
};
B::~B(){} // Error except in cfront mode
```

- When disambiguation requires deciding whether something is a parameter declaration or an argument expression, the pattern *type-name-or-keyword (identifier ...)* is treated as an argument. For example:

```
class A { A(); };
double d;
A x(int(d));
A(x2);
```

By default, `int(d)` is interpreted as a parameter declaration (with redundant parentheses), and so `x` is a function; but in `cfront` compatibility mode `int(d)` is an argument and `x` is a variable.

The declaration `A(x2)` is also misinterpreted by `cfront`. It should be interpreted as the declaration of an object named `x2`, but in `cfront` mode it is interpreted as a function style cast of `x2` to the type `A`.

Similarly, the following declaration declares a function named `xyz`, that takes a parameter of type function taking no arguments and returning an `int`. In `cfront` mode, this is interpreted as a declaration of an object that is initialized with the value `int()`, which evaluates to 0.

```
int xyz(int());
```

- A named bit field can have a size of 0. The declaration is treated as though no name had been declared.
- Plain bit fields (such as bit fields declared with a type of `int`) are always signed.

- The name given in an elaborated type specifier can be a typedef name that is the synonym for a class name. For example:

```
typedef class A T;
class T *pa;           // No error in cfront mode
```

- No warning is issued on duplicate size and sign specifiers, as shown in the following example:

```
short short int i;    // No warning in cfront mode
```

- Virtual function table pointer-update code is not generated in destructors for base classes of classes without virtual functions, even if the base class virtual functions might be overridden in a further derived class. For example:

```
struct A {
    virtual void f() {}
    A() {}
    ~A() {}
};
struct B : public A {
    B() {}
    ~B() {f();}           // Should call A::f according to ARM 12.7
};
struct C : public B {
    void f() {}
} c;
```

In cfront compatibility mode, B::~~B calls C::f.

- An extra comma is allowed after the last argument in an argument list. For example:

```
f(1, 2, );
```

- A constant pointer-to-member function can be cast to a pointer-to-function, as in the following example. A warning is issued.

```
struct A {int f();};
main () {
    int (*p)();
    p = (int (*)( ))A::f;           // Okay, with warning
}
```

- Arguments of class types that allow bitwise copy construction but also have destructors are passed by value like C structures, and the destructor is not called on the copy. In normal mode, the class object is copied into a temporary, the address of the temporary is passed as the argument, and the destructor is called on the temporary after the call returns. Because the argument is passed by value instead of by address, code like this compiled in `cfront` mode is not calling-sequence compatible with the same code compiled in normal mode. In practice, this is not much of a problem, since classes that allow bitwise copying usually do not have destructors.
- A union member may be declared to have the type of a class for which the user has defined an assignment operator (as long as the class has no constructor or destructor). A warning is issued.

- When an unnamed class appears in a `typedef` declaration, the `typedef` name may appear as the class name in an elaborated type specifier. For example:

```
typedef struct { int i, j; } S;
struct S x;           // No error in cfront mode
```

- Two member functions may be declared with the same parameter types when one is static and the other is nonstatic with a function qualifier. For example:

```
class A {
    void f(int) const;
    static void f(int); // No error in cfront mode
};
```

- The scope of a variable declared in the `for-init-statement` is the scope to which the `for` statement belongs. For example:

```
int f(int i) {
    for (int j = 0; j < i; ++j) { /* ... */ }
    return j; // No error in cfront mode
}
```

- Function types differing only in that one is declared `extern "C"` and the other `extern "C++"` can be treated as identical:

```
typedef void (*PF)();
extern "C" typedef void (*PCF)();
void f(PF);
void f(PCF);
```

By contrast, in standard C++, `PF` and `PCF` are different and incompatible types; `PF` is a pointer to an `extern "C++"` function whereas `PCF` is a pointer to an `extern "C"` function; and the two declarations of `f` create an overload set.

- Functions declared `inline` have internal linkage.
- `enum` types are regarded as integral types.

- An uninitialized const object of non-POD class type is allowed even if its default constructor is implicitly declared as in the following example:

```
struct A { virtual void f(); int i; };  
const A a;
```

- A function parameter type is allowed to involve a pointer or reference to array of unknown bounds.
- If the user declares an `operator=` function in a class, but not one that can serve as the default `operator=`, and bitwise assignment could be done on the class, a default `operator=` is not generated. Only the user-written `operator=` functions are considered for assignments, so bitwise assignment is not done.

# Using the Compiler Message System [C]

---

This appendix describes how to use the message system to control and use messages issued by the compiler. Explanatory texts for messages can be displayed online through the use of the `explain` command.

## C.1 Expanding Messages with the `explain` Command

You can use the `explain` command to display an explanation of any message issued by the compiler. The command takes as an argument, the message number, including the number's prefix. The prefix for Cray C and C++ is `CC`.

In the following sample dialog, the `cc` command invokes the compiler on source file `bug.c`. Message `CC-24` is displayed. The `explain` command displays the expanded explanation for this message.

```
% cc bug.c
CC-24 cc: ERROR File = bug.c, Line = 1
      An invalid octal constant is used.

      int i = 018;
              ^

1 error detected in the compilation of "bug.c".
% explain CC-24
```

An invalid octal constant is used.

Each digit of an octal constant must be between 0 and 7, inclusive. One or more digits in the indicated octal constant are outside of this range. Change each digit in the octal constant to be within the valid range.

## C.2 Controlling the Use of Messages

This section summarizes the command line options that affect the issuing of messages from the compiler.

## C.2.1 Command Line Options

Option	Description
<code>-h errorlimit[=<i>n</i>]</code>	Specifies the maximum number of error messages the compiler prints before it exits.
<code>-h [no]message=<i>n</i>[:...]</code>	Enables or disables the specified compiler messages, overriding <code>-h msglevel</code> .
<code>-h msglevel_<i>n</i></code>	Specifies the lowest severity level of messages to be issued.
<code>-h [no]msgs</code>	Enables or disables the writing of optimization messages to <code>stderr</code> .
<code>h [no]negmsgs</code>	Enables or disables the writing of messages to <code>stderr</code> that indicate why optimizations such as vectorization or inlining did not occur in a given instance.
<code>-h report=<i>args</i></code>	Generates optimization report messages.

## C.2.2 Environment Options for Messages

The following are used by the message system.

Variable	Description
NLSPATH	Specifies the default value of the message system search path environment variable.
LANG	Identifies your requirements for native language, local customs, and coded character set with regard to the message system.
MSG_FORMAT	Controls the format in which you receive error messages.

### C.2.3 ORIG\_CMD\_NAME Environment Variable

You can override the command name printed in the message. If the environment variable `ORIG_CMD_NAME` is set, the value of `ORIG_CMD_NAME` is used as the command name in the message. This functionality is provided for use with shell scripts that invoke the compiler. By setting `ORIG_CMD_NAME` to the name of the script, any message printed by the compiler appears as though it was generated by the script. For example, the following C shell script is named `newcc`:

```
#
setenv ORIG_CMD_NAME 'basename $0'
cc $*
```

A message generated by invoking `newcc` resembles the following:

```
CC-8 newcc: ERROR File = x.c, Line = 1
  A new-line character appears inside a string literal.
```

Because the environment variable `ORIG_CMD_NAME` is set to `newcc`, this appears as the command name instead of `cc(1)` in this message.



**Caution:** The `ORIG_CMD_NAME` environment variable is not part of the message system. It is supported by the Cray C and C++ compilers as an aid to programmers. Other products, such as the Fortran compiler and the loader, may support this variable. However, you should not rely on support for this variable in any other product.

You must be careful when setting the environment variable `ORIG_CMD_NAME`. If you set `ORIG_CMD_NAME` inadvertently, the compiler may generate messages with an incorrect command name. This may be particularly confusing if, for example, `ORIG_CMD_NAME` is set to `newcc` when the Fortran compiler prints a message. The Fortran message will look as though it came from `newcc`.

## C.3 Message Severity

Each message issued by the compiler falls into one of the following categories of messages, depending on the severity of the error condition encountered or the type of information being reported.

Category	Meaning
COMMENT	Inefficient programming practices.
NOTE	Unusual programming style or the use of outmoded statements.
CAUTION	Possible user error. Cautions are issued when the compiler detects a condition that may cause the program to abort or behave unpredictably.

Category	Meaning
WARNING	Probable user error. Indicates that the program will probably abort or behave unpredictably.
ERROR	Fatal error; that is, a serious error in the source code. No binary output is produced.
INTERNAL	Problems in the compilation process. Please report internal errors immediately to the system support staff, so a Software Problem Report (SPR) can be filed.
LIMIT	Compiler limits have been exceeded. Normally you can modify the source code or environment to avoid these errors. If limit errors cannot be resolved by such modifications, please report these errors to the system support staff, so that an SPR can be filed.
INFO	Useful additional information about the compiled program.
INLINE	Information about inline code expansion performed on the compiled code.
SCALAR	Information about scalar optimizations performed on the compiled code.
VECTOR	Information about vectorization optimizations performed on the compiled code.
OPTIMIZATION	Information about general optimizations.
IPA_INFO	Information about interprocedural optimizations.



## C.4 Common System Messages

The errors in the following list can occur during the execution of a user program. The operating system detects them and issues the appropriate message. These errors are not detected by the compiler and are not unique to C and C++ programs; they may occur in any application program written in any language.

- `Operand Range Error`

An operand range error occurs when a program attempts to load or store in an area of memory that is not part of the user's area. This usually occurs when an invalid pointer is dereferenced.

- `Program Range Error`

A program range error occurs when a program attempts to jump into an area of memory that is not part of the user's area. This may occur, for example, when a function in the program mistakenly overwrites the internal program stack. When this happens, the address of the function from which the function was called is lost. When the function attempts to return to the calling function, it jumps elsewhere instead.

- `Error Exit`

An error exit occurs when a program attempts to execute an invalid instruction. This error usually occurs when the program's code area has been mistakenly overwritten with words of data (for example, when the program stores in a location pointed to by an invalid pointer).



# Using Intrinsic Functions [D]

---

The C and C++ intrinsic functions either allow for direct access to some hardware instructions or result in generation of inline code to perform some specialized functions. These intrinsic functions are processed completely by the compiler. In many cases, the generated code is one or two instructions. These are called *functions* because they are invoked with the syntax of function calls.

To get access to the intrinsic functions, the Cray C++ compiler requires that either the `intrinsics.h` file be included or that the intrinsic functions that you want to call be explicitly declared. If the source code does not have an `intrinsics.h` statement and you cannot modify the code, you can use the `-h prototype_intrinsics` option instead. If you explicitly declare an intrinsic function, the declaration must agree with the documentation or the compiler treats the call as a call to a normal function, not the intrinsic function. The `-h nointrinsics` command line option causes the compiler to treat these calls as regular function calls and not as intrinsic function calls.

The types of the arguments to intrinsic functions are checked by the compiler, and if any of the arguments do not have the correct type, a warning message is issued and the call is treated as a normal call to an external function. If your intention was to call an external function with the same name as an intrinsic function, you should change the external function name. The names used for the Cray C intrinsic functions are in the name space reserved for the implementation.

**Note:** Several of these intrinsic functions have both a vector and a scalar version. If a vector version of an intrinsic function exists and the intrinsic is called within a vectorized loop, the compiler uses the vector version of the intrinsic. For details on whether it has a vector version, refer to the appropriate intrinsic function man page.

The following sections group the C and C++ intrinsics according to function and provides a brief description of each intrinsic in that group. For more information, see the corresponding man page.

## D.1 Bit Operations

The following intrinsic functions copy, count, or shift bits or computes the parity bit:

<code>_dshiftl</code>	Move the left most <i>n</i> bits of an integer into the right side of another integer, and return that integer.
<code>_dshiftr</code>	Move the right most <i>n</i> bits of an integer into the left side of another integer and return that integer.
<code>_pbit</code>	Copies the rightmost bit of a word to the <i>n</i> <sup>th</sup> bit, from the right, of another word.
<code>_pbits</code>	Copies the rightmost <i>m</i> bits of a word to another word beginning at bit <i>n</i> .
<code>_poppar</code>	Computes the parity bit for a variable.
<code>_popcnt</code>	
<code>_popcnt32</code>	
<code>_popcnt64</code>	Counts the number of set bits in 32-bit and 64-bit integer words.
<code>_leadz</code>	
<code>_leadz32</code>	
<code>_leadz64</code>	Counts the number of leading 0 bits in 32-bit and 64-bit integer words.
<code>_gbit</code>	<code>_gbit</code> returns the value of the <i>n</i> <sup>th</sup> bit from the right.
<code>_gbits</code>	Returns a value consisting of <i>m</i> bits extracted from a variable, beginning at <i>n</i> <sup>th</sup> bit from the right.

## D.2 Mask Operations

The following intrinsic functions create bit masks:

<code>_mask</code>	Creates a left-justified or right-justified bit mask with all bits set to 1.
<code>_maskl</code>	Returns a left-justified bit mask with <i>i</i> bits set to 1.
<code>_maskr</code>	Returns a right-justified bit mask with <i>i</i> bits set to 1.

## D.3 Miscellaneous Operations

The following intrinsic functions perform various functions:

`_int_mult_upper`

Multiplies integers and returns the uppermost bits. For more information, see the `int_mult_upper(3i)` man page.

`_ranf`

Computes a pseudo-random floating-point number ranging from 0.0 through 1.0.

`_rtc`

Return a real-time clock value expressed in clock ticks.



# Glossary

---

## **CLE**

The operating system for Cray XT systems.

## **CNL**

CNL is the Cray XT and Cray X2 compute node kernel. CNL provides a set of supported system calls. CNL provides many of the operating system functions available through the service nodes, although some functionality has been removed to improve performance and reduce memory usage by the system.

## **compute node**

A node that runs application programs. A compute node performs only computation; system services cannot run on compute nodes. Compute nodes run a specified kernel to support either scalar or vector applications. See also *node*; *service node*.

## **deferred implementation**

The label used to introduce information about a feature that will not be implemented until a later release.

## **login node**

The service node that provides a user interface and services for compiling and running applications.

## **node**

For CLE systems, the logical group of processor(s), memory, and network components acting as a network end point on the system interconnection network.

## **service node**

A node that performs support functions for applications and system services. Service nodes run SUSE LINUX and perform specialized functions. There are six types of predefined service nodes: login, IO, network, boot, database, and syslog.

**system interconnection network**

The high-speed network that handles all node-to-node data transfers.